AD-A274 369

(2)

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE NOTES





**FIRST EDITION**

**July 1993**

**by**

**Duane J. Jarc and Eric Fendler**

**in cooperation with**

**Undergraduate Programs**
**University of Maryland University College**

**Sponsored by Advanced Research Projects Agency (ARPA)**

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



**1. AGENCY USE ONLY** *(Leave blank)*

**2. REPORT DATE** July 1993

**3. REPORT TYPE AND DATES COVERED** LECTURE NOTES

**4. TITLE AND SUBTITLE**
CMSC 130 INTRODUCTORY COMPUTER SCIENCE
LECTURE NOTES

**5. FUNDING NUMBERS**
MDA972-92-J-1021

**6. AUTHOR(S)**
Duane J. Jarc & Eric Fendler

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
UNIVERSITY OF MARYLAND UNIVERSITY COLLEGE

**8. PERFORMING ORGANIZATION REPORT NUMBER**
CMSC 130

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
ARPA
3701 N. Fairfax Dr.
Arlington, VA 22203-1714

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**
APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

The CMSC 130 Introductory Computer Science lecture notes are used in the classroom for teaching CMSC 130, an introductory computer science course, using the Ada programming language.

**14. SUBJECT TERMS**
COMPUTER SCIENCE
ADA LANGUAGE
PROGRAMMING
LANGUAGE CONCEPTS
SOFTWARE CONCEPTS

**15. NUMBER OF PAGES** 169

**16. PRICE CODE**

**17. SECURITY CLASSIFICATION OF REPORT** U

**18. SECURITY CLASSIFICATION OF THIS PAGE** U

**19. SECURITY CLASSIFICATION OF ABSTRACT** U

**20. LIMITATION OF ABSTRACT** U / U

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102

First Edition—July 1993



This project is sponsored by ARPA. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

Printed in the United States of America.

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 1

# INTRODUCTION AND OVERVIEW

# OVERVIEW OF INTRODUCTORY COURSES

CMSC 130 Introductory Computer Science

- Basic Syntax And Semantics Of Ada Language
  - Conditional And Iterative Control Structures
  - Scalar Data Types, Arrays And Records
- Structured Programming Concepts

CMSC 135 Intermediate Computer Science

- Additional Ada Language Concepts
  - Generic Packages
  - Access Data Types
- Data Structures And Algorithm Efficiency
  - Stacks, Queues, Lists, Trees And Graphs
- Objected-Oriented Programming Concepts
- Recursive Programming Concepts

CMSC 230 Advanced Computer Science

- Advanced Ada Language Concepts
  - Task Declarations And Control Statements
- Concurrent Programming Concepts

# CMSC 130 OVERVIEW

Introductory Concepts

- Hardware/Software Concepts
- Language Concepts
- Software Engineering Concepts

Control Structures

- Conditional Control (If And Case Statements)
- Iterative Control (For, While And Loop Statements)
- Subprograms (Procedures And Functions)

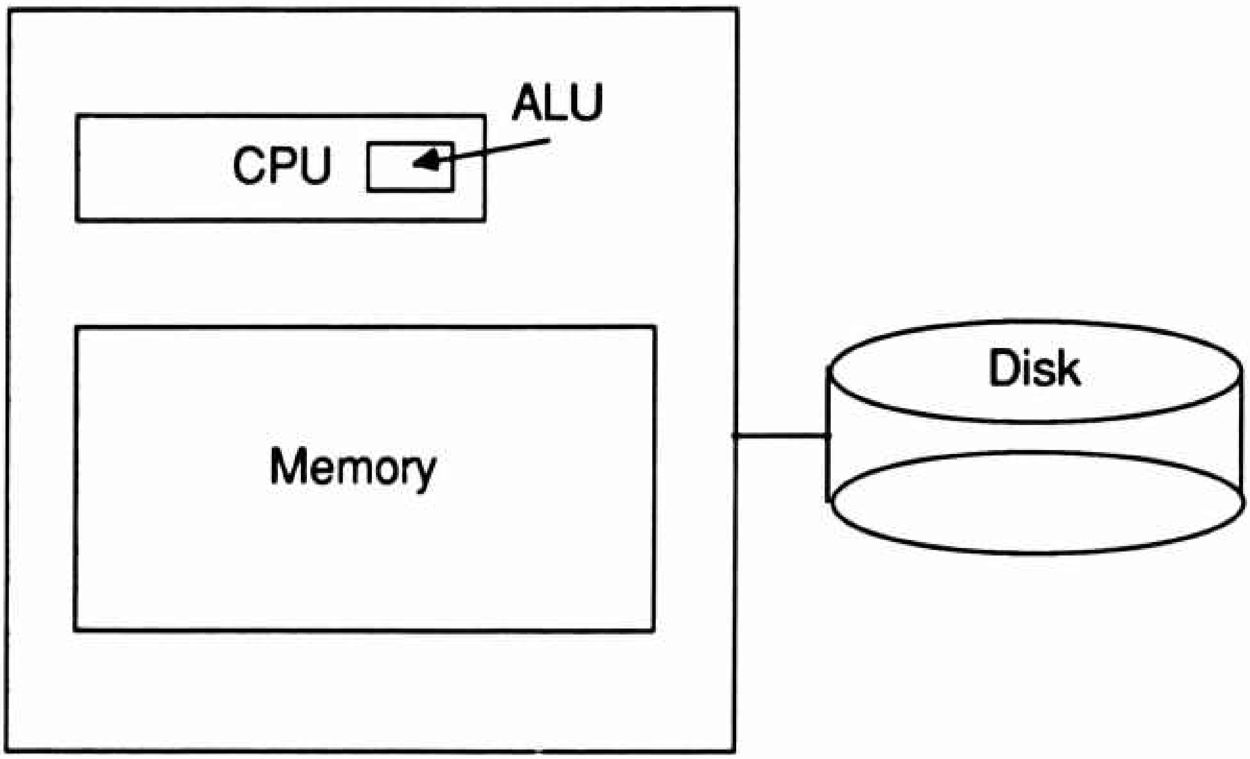
Data Structures

- Scalar Data Types
- Array Types
- Record Types

Abstraction Concept

- Procedural Abstraction
- Data Abstraction

# COMPUTER HARDWARE




Central Processing Unit (CPU)

Arithmetic Logic Unit (ALU)

Storage

Registers

Main Memory

Disk

Input/Output Devices

# COMPUTER SOFTWARE

System Software

- Operating System
  - Interface Between Computer Hardware And User
  - Manages System Resources

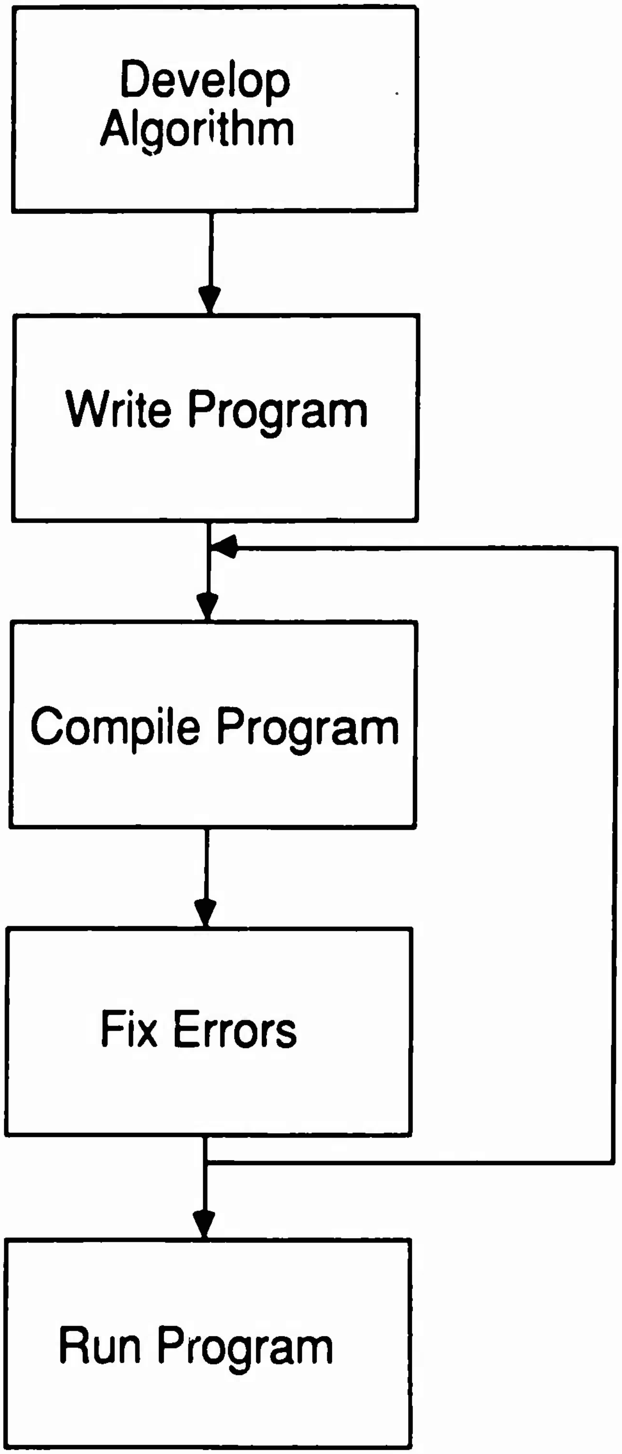
System Software (Program Development)

- Editor
  - Allows Text File Creation And Modification
- Compiler
  - Translates High Level Language Into Machine Language
- Linker
  - Links Separately Compiled Files Together
- Symbolic Debugger
  - Allows Program Tracing And Memory Examination
- Integrated Development Environment
  - Integrated Editor, Compiler, Linker And Debugger

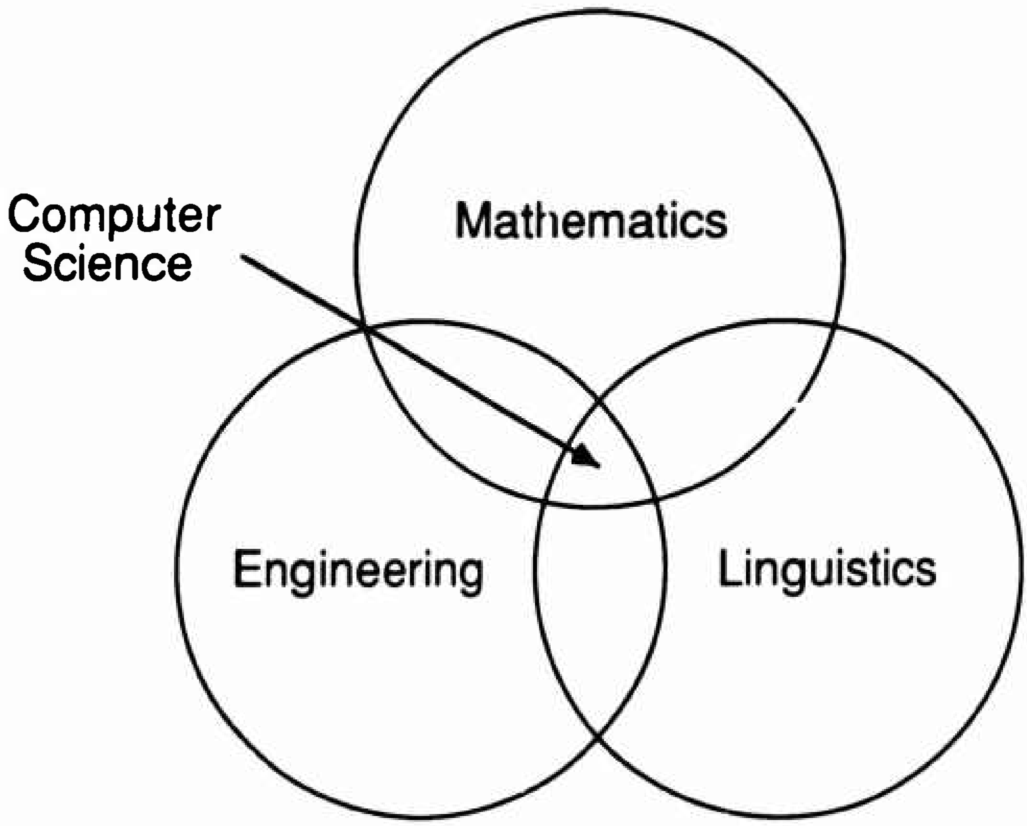
Application Software

- User Written Programs

# PROGRAM DEVELOPMENT PROCESS



# COMPUTER SCIENCE AS AN INTERDISCIPLINARY FIELD




Mathematical Component

Algorithms, Efficiency And Computability

Linguistic Component

Programming Languages

Engineering Component

Design Principles (Software Engineering)

# ALGORITHMS AND PROBLEM SOLVING . . . .

Terminology:

Algorithm: A Step By Step Procedure That Is Used To Solve A Problem

Sandwich Making Algorithm

1. Go To The Store And Buy Some Bread
2. Go Home And Put The Bread On A Plate
3. Go To The Store And Buy Some Mustard
4. Go Home And Put The Mustard On The Bread
5. Go To The Store And Buy Some Baloney
6. Go Home And Put The Baloney On The Bread
7. Eat The Sandwich

Important Points:

1. A Clear Problem Definition Must Precede Algorithm Development
2. Algorithm Development Determines The Details Of How To Solve The Problem
3. There Are Many Algorithms That Solve The Same Problem

# EFFICIENT ALGORITHMS

Terminology:

<u>Algorithm Efficiency</u>: A Measure Of The Number Of Steps Required To Complete An Algorithm

Efficient Sandwich Making Algorithm

1. Go To The Store And Buy Bread, Mustard And Baloney
2. Go Home And Put The Bread On The Plate
3. Put Mustard On The Bread
4. Put Baloney On The Bread
5. Eat The Sandwich

Important Points:

1. This Algorithm Is More Efficient Than The Previous Algorithm, It Save Two Steps, Two Trips To The Store
2. For More Complex Algorithms, The Number Of Steps Required May Not Be Constant, It May Depend On Some Input, The Type Of Sandwich, For Example
3. When The Number Of Steps Varies, Plotting A Graph Of The Steps Versus The Input Data Can Be Useful

# SOFTWARE ENGINEERING PRINCIPLES

Computer Science As An Engineering Discipline

1. Software Systems Require Design, An Essential Characteristic Of Engineering Fields
2. Computer Systems Are Constructed, The Building Material Is The Programming Language
3. Simplicity Is An Important Design Criterion, Simple Programs Are More Reliable And Easier To Maintain

Programming Languages And Software Engineering

1. Ada Is A Language Designed With Software Engineering Principles In Mind
2. Ada's Design Emphasizes Reliability Before Efficiency
3. Using A Well Designed Language Can Reduce The Cost Of Software Development

Ada Software Engineering Features

1. Structured Language With Fully Nested Syntax
2. Procedural Abstraction Supported By Subprograms
3. Data Abstraction Supported By Packages

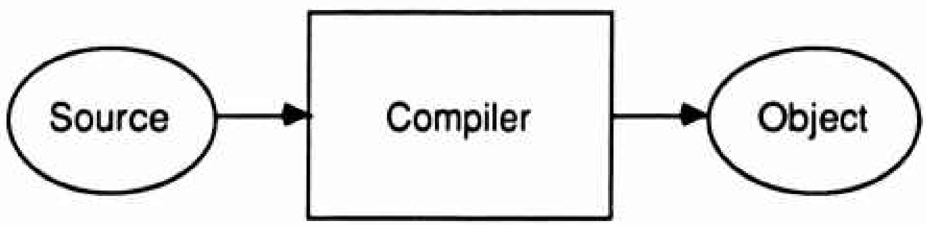
# PROGRAMMING LANGUAGES

## Characteristics Of Programming Languages

1. Programming Languages Are Formal Languages That Avoid The Potential Ambiguity Of Natural Language
2. Programming Languages Have A Precise Syntax, Programs With Syntax Errors Are Not Understood

## Languages And Compilers

1. Compilers Are Language Translators That Translate From A High-Level Language To Machine Language
2. Compilers Translate Complete Programs, Interpreters Translate One Line At A Time




## Abstraction Level Of Languages

1. Programming With Low-Level Languages Is Like Building With Small Bricks, More Work Is Involved
2. Programming With High-Level Languages Is Like Building With Prefabricated Panels, Less Work Is Involved

# EVOLUTION OF PROGRAMMING LANGUAGES

Unstructured Imperative Languages

- Machine Language
  - Physical (Numeric) Addresses, Numeric Operation Codes
- Assembly Language
  - Symbolic Addresses And Operation Codes

Structured Imperative Languages

- Languages With Arithmetic Formulas (Fortran)
  - Nested Expressions
- Fully Structured Languages (Pascal, C, Ada)
  - Nested Statements, No Gotos Required

Nonimperative Languages

- Functional Programming Languages (ML)
  - No States, No Variables, No Assignments
- Logic Programming Languages (Prolog)
  - No Explicit Control Flow, Nonprocedural

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 2

# SIMPLE ADA PROGRAMS

# ADA HISTORY

Developed By DoD In Response To High Cost Of Software

Ada Development Was A Ten Year Effort

1973-1974 Cost Study To Find Ways To Reduce Software Costs

1975 High Order Language Working Group (HOLWG)

Goal To Adopt Standard DoD Language

Developed Requirement Specifications

(Strawman, Woodenman, Tinman)

1977 Concluded No Existing Language Met Specifications

Solicited Proposals For New Language

17 Proposals Received, All Based On Pascal Excepts IBM's

4 Selected, Designated Red, Blue, Green and Yellow

1978 Two Finalists selected

Red (Intermetrics), Green (CII Honeywell Bull)

1979 Green Language Chosen

1981-1982 Extensive Public Reviews

1983 Military Standard & ANSI Standard Adopted

First Compilers Available In Mid 1980's

# LEVELS OF LANGUAGE

| Natural Language | Programming Language |
|---|---|
| Letters | Characters |
| Words | Tokens |
| Phrases | Expressions |
| Sentences | Statements |
| Paragraphs | Subprograms |
| Sections | Programs |

# LEXICAL COMPONENTS

Natural Language

- Parts Of Speech
  - Noun, Verb, Adjective, Adverb, Preposition

Programming Language

- Token Categories
  - Reserved Words, Identifiers, Constants, Operators

# IDENTIFIERS AND RESERVED WORDS

## Identifier Syntax

Identifiers Must Begin With A Letter, Followed By Zero Or More Letters, Digits Or Underscores, Consecutive Underscores And Terminating Underscores Are Prohibited

## Valid Identifiers

`Name` `This_Value` `Value_2`

## Invalid Identifiers

| | |
|---|---|
| `5_Value` | Begins With Digit |
| `Some__Number` | Consecutive Underscores |
| `Statement#` | Contains Special Character |
| `Identifier_` | Terminating Underscore |

## Role Of Identifiers

Identifiers Are Used To Name Variables, Constants Etc.

Unlike English Words Which Are Fixed, Programmers Can Create Any Identifier That Follows The Syntax Rules And Give It A Meaning

## Reserved Words

Certain Ada Identifiers, If Case While And Others, Are Reserved, Have A Predefined Meaning, They Can Not Be User-Defined Identifiers

# LEXICAL STYLE

## Comments

Comments Add Clarification To Programs, They Begin With Two Dashes And End With End Of Line

```
-- This Is An Ada Comment
```

## Upper And Lower Case Issues

Ada Is Case Insensitive, Upper And Lower Case Characters Are Not Distinct

There Is No Widely Observed Convention For The Use Of Upper And Lower Case

## Upper And Lower Case Convention Adopted

Upper Case For Reserved Words

Mixed Case For User Defined Identifiers

## Blank Space

Any Number Of Spaces Can Separate Tokens

Blank Space Should Be Used Liberally To Enhance The Readability Of Programs

## Indentation

Indentation With Blank Space Should Be Used To Reflect The Control Structure Of The Program

# DATA TYPES

## Data Type Concept

Data Types In A Programming Language Categorize Data In A Similar Way That Categories Such As Animate, Inanimate Categorize Objects In Natural Language

## Character Data Type

Literal Representation: A Single Character Enclosed In A Pair Of Single Quotes

Literal Examples: `'A'` `'b'`

## Integer Data Type

Literal Representation: An Optionally Signed Sequence Of Digits With An Optional Exponent

Literal Examples: `156` `-320` `12E2`

## Float Data Type

Literal Representation: An Optionally Signed Sequence Of Digits Followed By A Decimal Point, Another Sequence Of Digits And An Optional Exponent

Literal Examples: `98.6` `2.45E-2` `-0.4`

Invalid Examples

| | |
|---|---|
| `2.E2` | Must Be A Digit After The Decimal |
| `.21` | Must Be A Digit Before The Decimal |

# ARITHMETIC EXPRESSIONS

## Role Of Arithmetic Expressions

Arithmetic Expressions Define Mathematical Formulas

Arithmetic Expressions Contain Operators And Operands (Literals Or Identifiers Naming Variables Or Constants)

## Arithmetic Operators

| | |
|---|---|
| `+` | Addition |
| `-` | Subtraction |
| `*` | Multiplication |
| `/` | Division |
| `REM` | Remainder |
| `**` | Exponentiation |

## Arithmetic Expression Examples

```
Integer_Variable + 5
```

```
Number * (Value + 2)
```

## Expression Evaluation

Parentheses Can Be Used To Group Subexpressions

In The Absence Of Parentheses Precedence Applies

Highest Precedence `**`
Middle Precedence `*  /  REM`
Lowest Precedence `+  -`

Left To Right Associativity Applies Otherwise

# DECLARATIVE STATEMENTS

## Variable Declarations

A Variable Declaration Instructs The Compiler To Reserve A Memory Location For A Variable Of The Specified Type

Identifiers Are Used To Name Variables

## Variable Declaration Examples

```
Letter: Character;
Whole_Number: Integer;
Real_Number1,Another_Real: Float;
```

## Variable Declaration Syntax

variable_declaration ::=
    identifier_list : type;

## Constant Declarations

A Constant Declaration Instructs The Compiler To Reserve Memory For A Value That Can Not Change

## Constant Declaration Examples

```
Excellent_Grade: CONSTANT Character := 'A';
Course_Number: CONSTANT Integer := 130;
```

## Constant Declaration Syntax

constant_declaration ::=
    identifier : `CONSTANT` type := value;

# EXECUTABLE STATEMENTS

## Assignment Statements

An Assignment Statement Stores The Value Of The Expression On The Right Side Of The Assignment Into The Variable On The Left Hand Side

## Assignment Statement Examples

```
Letter := 'B';
Real_Number := Another_Real + 5.0;
```

## Assignment Statement Syntax

assignment_statement ::=
    variable := expression;

## Input/Output Statements

An Input Statement Reads Data In From The Keyboard, An Output Statement Writes Data Out To The Screen

## Input/Output Statement Examples

```
Get(Whole_Number);
Put(Real_Number);
```

## Input/Output Statement Syntax

input_statement ::=
    `Get` (variable);

output_statement ::=
    `Put` (variable);

# COMPLETE PROGRAM SYNTAX

Simple Procedure Syntax

ada_program ::=
```
    WITH Text_IO;
    PROCEDURE identifier IS
       declarations
       i/o_package_instantiations
    BEGIN
       statements
    END identifier;
```

Important Points:

1. The Simplest Ada Program Consists Of One Procedure

2. Both Of The Identifiers, Which Name The Procedure, Must Match

3. Input/Output Package Instantiation Is Required When Integer And Floating Point I/O Is Required

Input/Output Package Instantiation

integer_io_package_instantiation ::=
```
    PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
```

floating_point_io_package_instantiation ::=
```
    PACKAGE Flt_IO IS NEW Text_IO.Float_IO
    (Float);
```

# ADA PROGRAM EXAMPLES

## Hello World Program

```
WITH Text_IO;
PROCEDURE Hello_World IS
  -- Declarative Section
BEGIN
  --Sequence of Statements
  Text_IO.Put_Line("Hello, World!!!");
END Hello_World;
```

## Adding Machine Program

```
WITH Text_IO;
PROCEDURE Adding_Machine IS
  Total: Integer;
  Users_Entry: Integer;
  One: CONSTANT Integer := 1;
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
BEGIN
  Text_IO.Put_Line("Enter an integer");
  Int_IO.Get(Users_Entry);
  Total := Users_Entry + One;
  Text_IO.Put("The answer is: ");
  Int_IO.Put(Total);
  Text_IO.New_Line;
END Adding_Machine;
```

# ERROR MESSAGES

Compilation Errors

These Are Errors Detected By The Compiler When The Program Is Compiled

Syntax Errors:

Misspelling Reserved Words Or Omitting Punctuation Are Examples

Semantic Errors:

Mismatched Types Are Examples

Run-Time Errors

These Are Errors Detected When The Program Is Run

Logic Errors

These Are Errors That Do Not Generate Error Messages, They Can Only Be Detected By Observing That Programs Generate Incorrect Output

Important Points:

1. Ada Is Designed To Encourage Early Error Detection

2. It Is Easier To Find And Correct Compilation Errors Than It Is To Detect Logic Errors

3. Symbolic Debuggers Can Be A Useful Tool To Uncover Logic Errors

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 3

# SOFTWARE ENGINEERING CONCEPTS

# SOFTWARE ENGINEERING

Major Issue:

Software Engineering Principles Promote Simple Program Design

Structured Programming

Structured Programming Simplifies The Statement Level Control Structure Of Programs By Prohibiting Explicit Gotos

Procedural Abstraction

Procedural Abstraction Simplifies Programs By Limiting The Size Of Subprograms

Data Abstraction

Data Abstraction Simplifies Programs By Encapsulating Data Type Definitions Into Packages

Ada And Software Engineering

Ada Supports Structured Programming Because It Contains High Level Statements That Can Be Nested

Ada Supports Procedural Abstraction Because Ada Permits The Definition And Invocation Of Subprograms

Ada Supports Data Encapsulation Because Ada Permits Data Types And Objects To Be Defined Within Packages

# STRUCTURED PROGRAMMING

Terminology:

<u>Flow Chart</u>: A Diagram That Illustrates The Flow Of Control Of A Computer Program

Unstructured Programs

Goto Statements Implement Control Flow

Flow Charts Can Contain Crossing Lines

Structured Programs

High-Level Statements Implement Control Flow

Flow Charts Never Contain Crossing Lines

Structured Program Control

Fundamental Statements

Simple Statements, Assignment I/O
Conditional Statements If, Case
Iterative Statements For, While And Loop

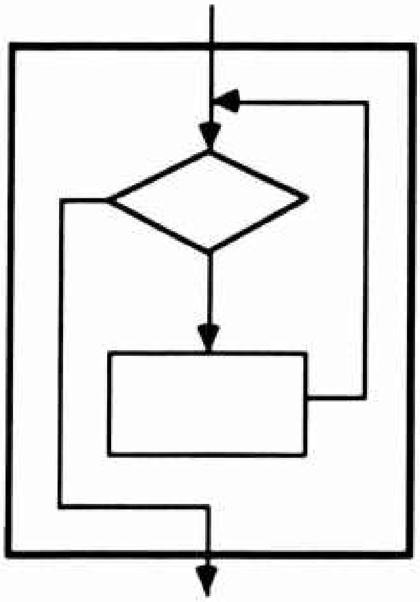
Methods Of Combining Statements

Sequential
Nested

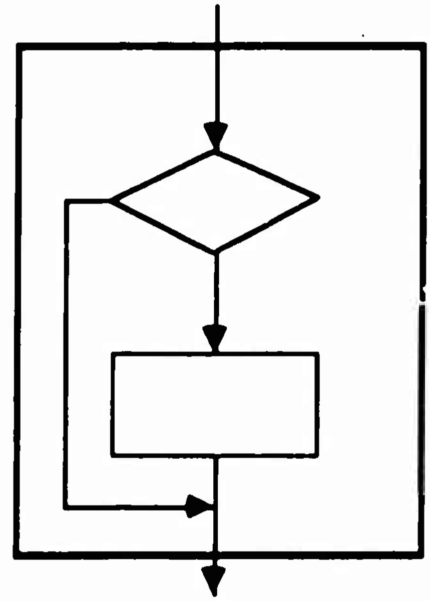
Important Point:

1. Nesting Is Essential To Structured Programming

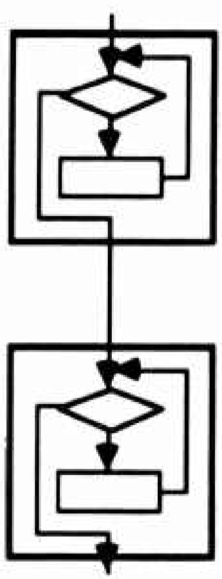
# BUILDING STRUCTURED PROGRAMS



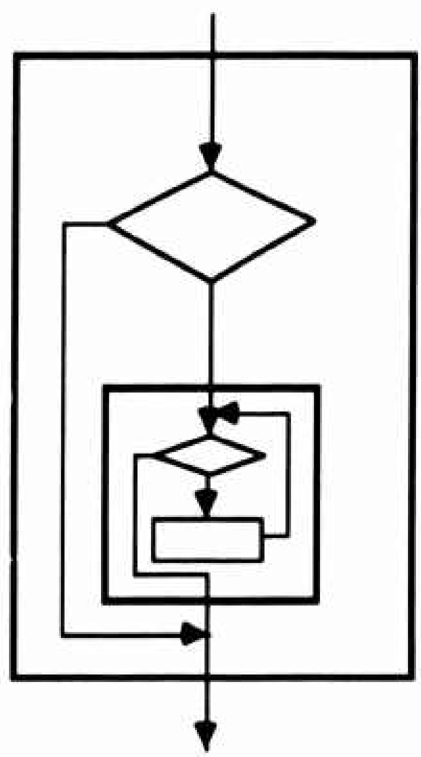
ITERATIVE CONTROL



CONDITIONAL CONTROL



SEQUENTIAL COMBINATION



NESTED COMBINATION

# UNSTRUCTURED CODE EXAMPLE

Unstructured Name Repetition Algorithm

1. Output "What's your name"
2. Input Name
3. Output "How many times should I print your name"
4. Input Repetitions
5. IF Repetitions > 100 GOTO Step 10
6. Output "Your name is ", Name
7. Decrement Repetitions
8. IF Repetitions Is Zero GOTO Step 12
9. GOTO Step 6
10. Output "That's too many times, tell me again"
11. GOTO Step 4
12. Output "Done"

Important Points:

1. The Code Is Hard To Read

2. The Steps In The Algorithm Are Tangled Up With One Another

3. The Code Is Really Delicate, It Is Difficult To Add New Functions Without Creating Problems

4. Flowcharts Were Traditionally Used For The Design Of Unstructured Code

# STRUCTURED CODE EXAMPLE

Structured Name Repetition Algorithm

```
Output "What's your name"
Input Name
Set Undetermined To True
WHILE Undetermined LOOP
    Output "How many times should I print your name"
    Input Repetitions
    IF Repetitions > 100 THEN
        Output "That's too many times, tell me again"
    ELSE
        Set Undetermined To False
    END IF
END LOOP
FOR Index IN 1..Repetitions LOOP
    Output "Your name is ", Name
END LOOP
Output "Done"
```

Important Points:

1. Structured Code Is Naturally Indented
2. The Indentation Of Structured Code Reflects The Control Structure
3. Structured Code Is Easier To Read And Easier To Modify

# PROCEDURAL ABSTRACTION

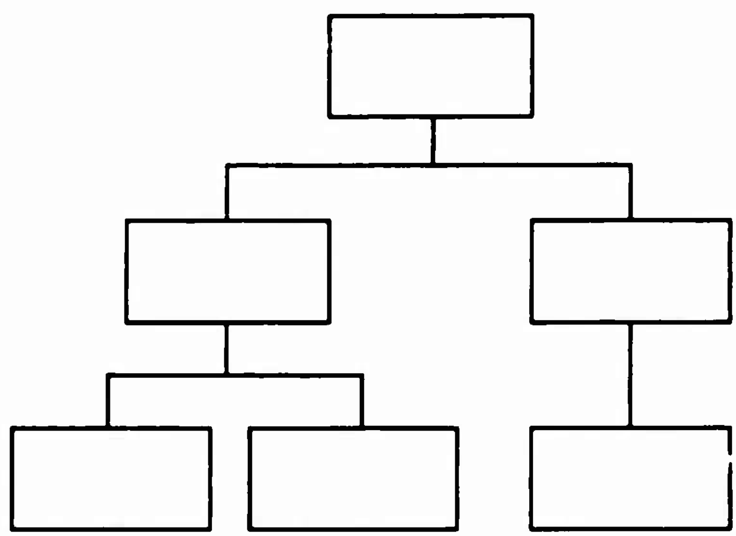
Terminology:

Top-down Design: A Method Of Program Design That Begins At The Top, By Subdividing The Whole Problem

Step-wise Refinement: The Process Of Further Subdividing, Refinement, The Problem Design At Each Step

Structure Chart: A Chart Representing The Functional Decomposition Of A Problem

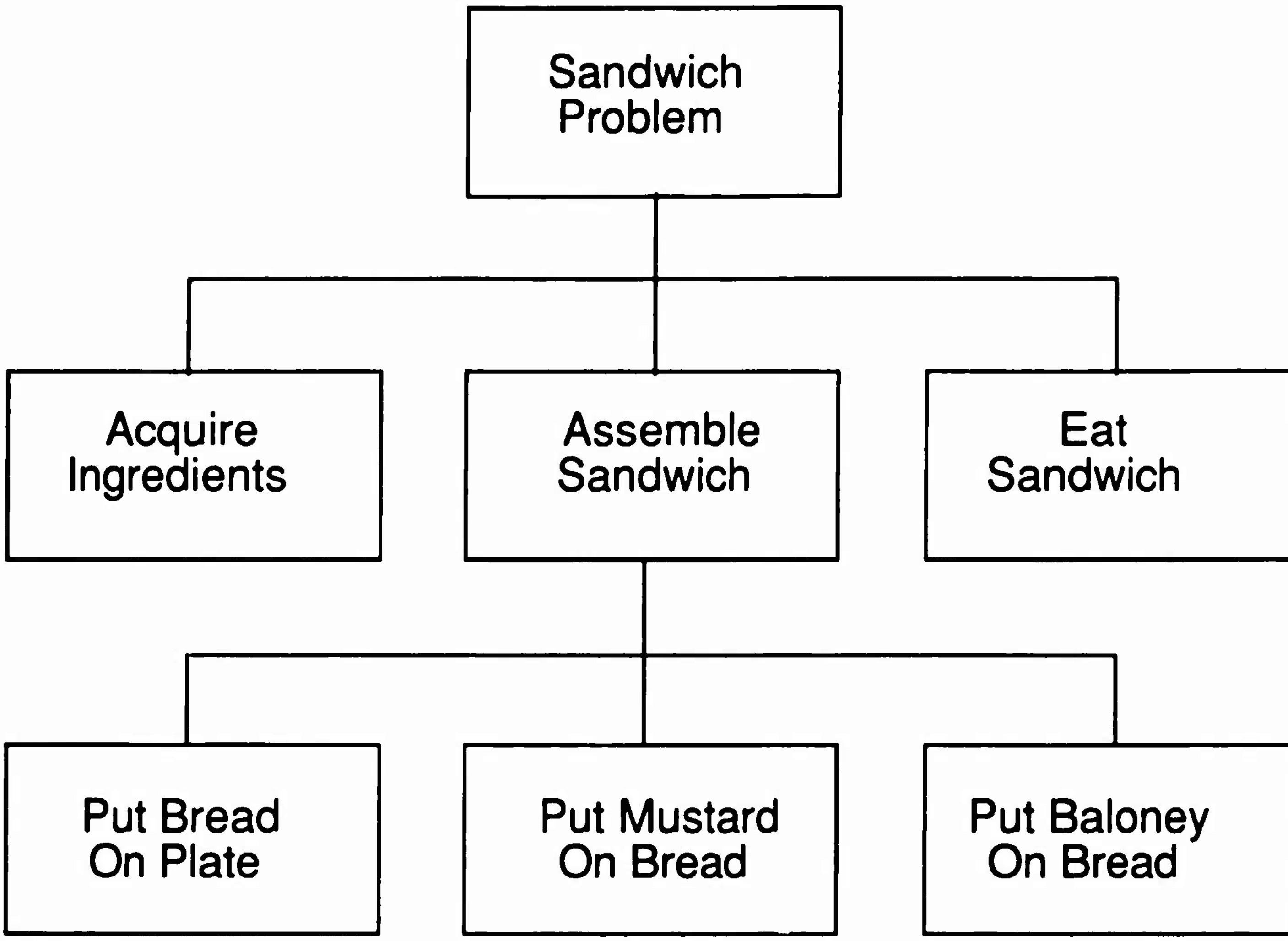
Structure Chart Example:



Important Points:

1. The Boxes Of A Functional Decomposition Most Often Become Subprograms
2. Get And Put Are Examples Of System Defined Procedural Abstraction

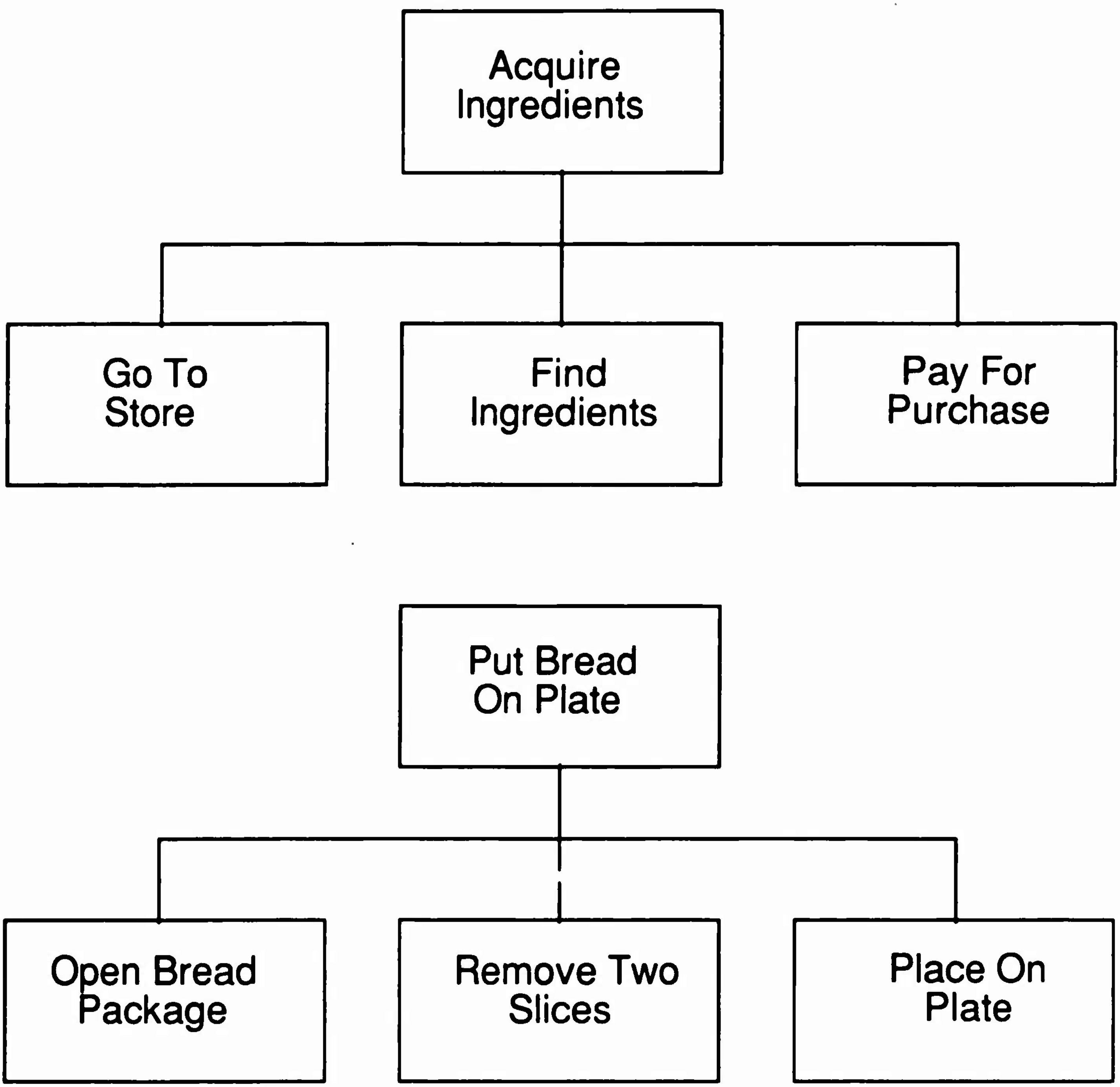
# TOP-DOWN DESIGN EXAMPLE




Important Point:

1. Computers Only Do What They Are Told, Nothing Is Obvious, Every Step That Must Be Performed Must Be Specified

# STEP-WISE REFINEMENT EXAMPLE



# DATA ABSTRACTION

Terminology:

Encapsulation: Enclosing A Data Type Definition With The Functions That Define Its Operations

Information Hiding: Concealing The Representation Of A Data Type And The Implement Of Its Operations

Loose Coupling: Building A System Of Software Components With Minimal Interdependencies

Software Reusability: Building General Software Components That Can Be Used In Other Systems

Ada Package Features

An Ada Package Encapsulations A Data Type Definition Together With The Functions That Define Its Operations

Ada Packages Consist Of Specifications And Bodies, The Package Body Conceals The Implementation Of The Operations

Package Specifications Consist Of Public And Private Parts, The Private Part Conceals The Representation Of Data Types

The Package Specification Defines The Interface Of A Package And Defines It Coupling With Other Units

Ada Packages Are Reusable Software Components That Can Be Separately Compiled

# ENUMERATION TYPES

## Enumeration Type Declarations

Enumeration Type Declarations Create New Data Types Whose Literal Values Are Names, Identifiers

## Enumeration Type Declaration Examples

```
TYPE Days IS (Monday, Tuesday, Wednesday,
   Thursday, Friday, Saturday, Sunday);

TYPE Rainbow_Colors IS (Red, Orange, Yellow,
   Green, Blue, Indigo, Violet);
```

## Enumeration Type Declaration Syntax

enumeration_type_declaration ::=
  `TYPE` identifier ( identifier_list ) ;

## Terminology:

Overloading: The Use Of A Name Or An Operator For More Than One Purpose

## Overloaded Enumeration Literals

Ada Permits Enumeration Literals To Be Overloaded, In Addition To `Rainbow_Colors`, The Following Type Could Be Defined

```
TYPE Primary_Colors IS (Red, Blue, Yellow);
```

The Names Red, Blue And Yellow Belong To Two Types

# ENUMERATION TYPE ATTRIBUTES & I/O

Terminology:

<u>Attribute</u>: A Constant Value Or A Function Associated With A Data Type

Attributes

| | |
|---|---|
| `First` | First Value Of The Type |
| `Last` | Last Value Of The Type |
| `Pos` | A Function That Maps Enumeration Literals To Their Position In The Type Definition |
| `Val` | A Function That Maps Positions To Enumeration Literals |
| `Succ` | Successor Function |
| `Pred` | Predecessor Function |

Attribute Examples

```
Days'First = Monday
Rainbow_Colors'Pos(Orange) = 1
Primary_Colors'Succ(Red) = Blue
```

Enumeration I/O Syntax:

enumeration_io ::=
`PACKAGE` identifier `IS NEW Text_IO.`
`Enumeration_IO(`type`);`

# ENUMERATION TYPE EXAMPLE

## Yesterday And Tomorrow Program

```
WITH Text_IO;
PROCEDURE Yesterday_And_Tomorrow IS
  TYPE Days IS (Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday);
  PACKAGE Days_IO IS NEW Text_IO.
    Enumeration_IO(Days);
  Today, Yesterday, Tomorrow: Days;
BEGIN
  Text_IO.Put_Line("What Day Is Today?");
  Days_IO.Get(Today);
  Yesterday := Days'Pred(Today);
  Tomorrow := Days'Succ(Today);
  Text_IO.Put("Yesterday was ");
  Days_IO.Put(Yesterday);
  Text_IO.New_Line;
  Text_IO.Put("Tomorrow will be ");
  Days_IO.Put(Tomorrow);
  Text_IO.New_Line;
END Yesterday_And_Tomorrow;
```

## Important Point:

1. A Run-Time Error Will Occur If Monday Or Sunday Is Entered, Monday Has No Predecessor, Sunday Has No Successor

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 4

# CONDITIONAL CONTROL

# SIMPLE CONDITIONS

## Boolean Data Type

The Boolean Data Type Is A Predefined Enumeration Type

```
TYPE Boolean IS (False,True);
```

## Relational Operators

| Symbol | Meaning |
|---|---|
| < | Less Than |
| <= | Less Than Or Equal |
| = | Equal |
| /= | Not Equal |
| > | Greater Than |
| >= | Greater Than Or Equal |

## Condition Syntax

condition ::=
operand relational_operator operand

## Important Point:

1. The Operands Of Simple Conditions Are Either Variables, Named Constants Or Literal Constants

# SIMPLE CONDITION EXAMPLES

## Type Checking Rule

The Type Of The Two Operands Of A Condition Must Match, They Must Be The Same Type

## Variable Declarations

```
Whole_Number: Integer := 1;

Decimal_Number: CONSTANT Float := 1.0;

Truth_Value: Boolean;

Letter: Character;
```

## Examples Of Valid Conditions

```
Whole_Number < 5

Truth_Value = True

Letter > 'A'
```

## Examples Of Invalid Conditions

```
Whole_Number < Decimal_Number

Truth_Value = 5

Decimal_Number > 2

Letter = Whole_Number
```

# SIMPLE IF STATEMENT

## Simple If Statement Syntax

```
simple_if_statement ::=
    IF condition THEN
      sequence_of_statements
    END IF;
```

## Absolute Values Procedure

```
WITH Text_IO;
PROCEDURE Absolute_Values IS
  Absolute, Number: Integer;
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
BEGIN
  Text_IO.Put("Enter An Integer: ");
  Int_IO.Get(Number);
  Absolute := ABS Number;
  Text_IO.Put("Absolute Value With ABS ");
  Int_IO.Put(Absolute);
  Text_IO.New_Line;
  IF Number < 0 THEN
     Number := -Number;
  END IF;
  Text_IO.Put("Absolute Value With IF ");
  Int_IO.Put(Number);
  Text_IO.New_Line;
END Absolute_Values;
```

# IF STATEMENT WITH ELSE CLAUSE

## If Else Statement Syntax

```
if_else_statement ::=
    IF condition THEN
      sequence_of_statements
    ELSE
      sequence_of_statements
    END IF;
```

## Circular Tomorrow Procedure

```
WITH Text_IO;
PROCEDURE Circular_Tomorrow IS
  TYPE Days IS (Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday);
  PACKAGE Days_IO IS NEW Text_IO.
    Enumeration_IO(Days);
  Today, Tomorrow: Days;
BEGIN
  Text_IO.Put_Line("What Day Is Today?");
  Days_IO.Get(Today);
  IF Today /= Sunday THEN
    Tomorrow := Days'Succ(Today);
  ELSE
    Tomorrow := Monday;
  END IF;
  Text_IO.Put("Tomorrow will be ");
  Days_IO.Put(Tomorrow); Text_IO.New_Line;
END Circular_Tomorrow;
```

# DESCENDING LOOP EXAMPLE

## Rocket Launch Program

```
WITH Text_IO;
PROCEDURE Rocket_Launch IS
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
BEGIN
  FOR Launch_Count IN REVERSE 1..3 LOOP
    Int_IO.Put(Launch_Count);
    IF Launch_Count = 3 THEN
      Text_IO.Put_Line
        ("Start launch computers");
    ELSIF Launch_Count = 2 THEN
      Text_IO.Put_Line
        ("Release rocket stabilizers");
    ELSIF Launch_Count = 1 THEN
      Text_IO.Put_Line
        ("Start engine ignition");
    END IF;
  END LOOP;
  Text_IO.Put_Line("Blast-Off");
END Rocket_Launch;
```

## Important Point:

1. In A Descending For Statement The Range Of Values Is Specified In Ascending Order

# ENUMERATION RANGE EXAMPLE

## Total Weeks Earnings Program

```
WITH Text_IO;
PROCEDURE Total_Weeks_Earnings IS
  TYPE Days_Of_Week IS (Sun, Mon, Tue, Wed,
    Thu, Fri, Sat);
  Weeks_Income, Todays_Income: Float := 0.0;
  PACKAGE Day_IO IS NEW
    Text_IO.Enumeration_IO(Days_Of_Week);
  PACKAGE Income_IO IS NEW Text_IO.Float_IO
    (Float);
BEGIN
  FOR Day IN Mon..Fri LOOP
    Text_IO.New_Line;
    Text_IO.Put("Enter salary & tips for ");
    Day_IO.Put(Day);
    Income_IO.Get(Todays_Income);
    Weeks_Income := Weeks_Income +
      Todays_Income;
  END LOOP;
  Text_IO.New_Line(3);
  Text_IO.Put("Total for the week is " );
  Income_IO.Put(Weeks_Income);
  Text_IO.New_Line;
END Total_Weeks_Earnings;
```

# NESTED LOOP EXAMPLE

## Oldest Go First Program

```
WITH Text_IO;
PROCEDURE Oldest_Go_First IS
  TYPE Months IS (Jan, Feb, Mar, Apr, May,
    Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  PACKAGE Age_IO IS NEW Text_IO.Integer_IO
    (Integer);
  PACKAGE Months_IO IS NEW
    Text_IO.Enumeration_IO(Months);
BEGIN
  FOR Age IN REVERSE 0..100 LOOP
    FOR Month_Born IN Jan..Dec LOOP
      Text_IO.Put("Now Serving: ");
      Age_IO.Put(Age);
      Text_IO.Put(" year olds born in ");
      Months_IO.Put(Month_Born);
      Text_IO.New_Line;
    END LOOP;
  END LOOP;
END Oldest_Go_First;
```

## Important Point:

1. The Statements In The Body Of The Innermost Loop Are Executed 1212 Times

   12 (Months) $\times$ 101 (Ages) = 1212 (Iterations)

# COUNTING EXECUTED STATEMENTS

Major Issues:

1. There Is Always An Upper Bound To The Number Of Statements Executed By Programs Containing Only Conditional Control Statements

2. The Number Of Statements Executed By Programs Contains Definite Iteration May Be Unbounded

Triangular Numbers Program

```
BEGIN                                          1
  Text_IO.Put("Enter Number: ");               1
  Int_IO.Get(Number);                          1
  Triangle := 0;                               1
  FOR Index IN 1..Number LOOP                  n
    Triangle := Triangle + Index;              n
  END LOOP;                                    n
  Text_IO.Put("Triangular Number = "); 1
  Int_IO.Put(Triangle);                        1
END Triangular_Numbers;                        1
```

Function Determining Executed Statements:

$f(n) = 3n + 7$

Important Point:

1. The Number Of Statements Executed Is Unbounded Because It Depends On The Input Value For Number

# ALGORITHM EFFICIENCY

Linear Efficiency:

Programs Containing Single Loops Have Linear Efficiency, The Function That Measures The Executed Statements Is Of The Form:

$$f(n) = an + b$$

Quadratic Efficiency

Programs Containing Nested Loops As Follows:

```
FOR I IN 1..n LOOP
  FOR J IN 1..n LOOP
```

Have Quadratic Efficiency, The Function That Measures The Executed Statements Is Of The Form:

$$f(n) = an^2 + bn + c$$

Comparing Algorithms:

Algorithms With Quadratic Execution Functions Are Less Efficient Than Algorithms With Linear Execution Functions

Important Points:

1. All Problems Can Be Solved With More Than One Algorithm

2. When The Efficiency Of The Algorithms For A Given Problem Vary, The Most Efficient Is Preferred

# SUBTYPE DECLARATIONS

## Subtype Concept

The Rationale For Using Subtype Declarations Is To Limit The Range Of Values And Detect Out Of Range Conditions If They Should Occur

## Subtype Declaration Syntax

subtype_declaration ::=
  `SUBTYPE` identifier `IS` type `RANGE`
  simple_expression . .simple_expression ;

## Subtype Declaration Examples

```
SUBTYPE Days_In_Month IS Integer RANGE
  1..31;

SUBTYPE Weekdays IS Days_Of_Week RANGE
  Monday..Friday;
```

## Range Checking Relational Operators

`IN` Determines Whether A Value Is In A Range

`NOT IN` Determines Whether A Value Is Not In A Range

## Range Checking Expression Examples

| | |
|---|---|
| `3 IN 5..10` | False |
| `Wednesday IN Monday..Friday` | True |
| `40 NOT IN 1..10` | True |

# SUBTYPE PROGRAM EXAMPLE

## Summer Only Program

```
WITH Text_IO;
PROCEDURE Summer_Only IS
  TYPE Months IS (Jan, Feb, Mar, Apr, May,
    Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  SUBTYPE Summer_Months IS Months RANGE
    Jun..Aug ;
  PACKAGE Age_IO IS NEW Text_IO.Integer_IO
    (Integer);
  PACKAGE Months_IO IS NEW
    Text_IO.Enumeration_IO(Months);
BEGIN
  FOR Age IN REVERSE 0..100 LOOP
    FOR Month_Born IN Jan..Dec LOOP
      IF Month_Born IN Summer_Months THEN
        Text_IO.Put("Now Serving: ");
          Age_IO.Put(Age);
        Text_IO.Put( " year olds born in " &
          "the summer month of " );
        Months_IO.Put(Month_Born);
        Text_IO.New_Line ;
      END IF;
    END LOOP;
  END LOOP;
END Summer_Only;
```

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 7

# INDEFINITE ITERATION AND PROCEDURES

# INDEFINITE ITERATION CONCEPT

Major Issues:

1. Programs Containing Only Conditional Control And Definite Iteration Always Terminate

2. Certain Problems Require Indefinite Iterative Control To Be Solved

Terminology:

<u>Indefinite Iteration</u>: Control Mechanism That Permits The Repetition Of A Group Of Statements While A Specified Condition Remains True

Important Points:

1. Programs Containing Indefinite Iteration Can Get Into "Infinite Loops" And Never Terminate

2. Indefinite Iteration Can Be Used To Solve Any Problem Solved With Definite Iteration

A Simple Problem Requiring Indefinite Iteration
Summing Positive Integers

1. Read In Integers From The Keyboard And Sum The Numbers

2. Stop When A Number That Is Not Positive Is Read In

3. Print Out The Sum

# WHILE STATEMENT

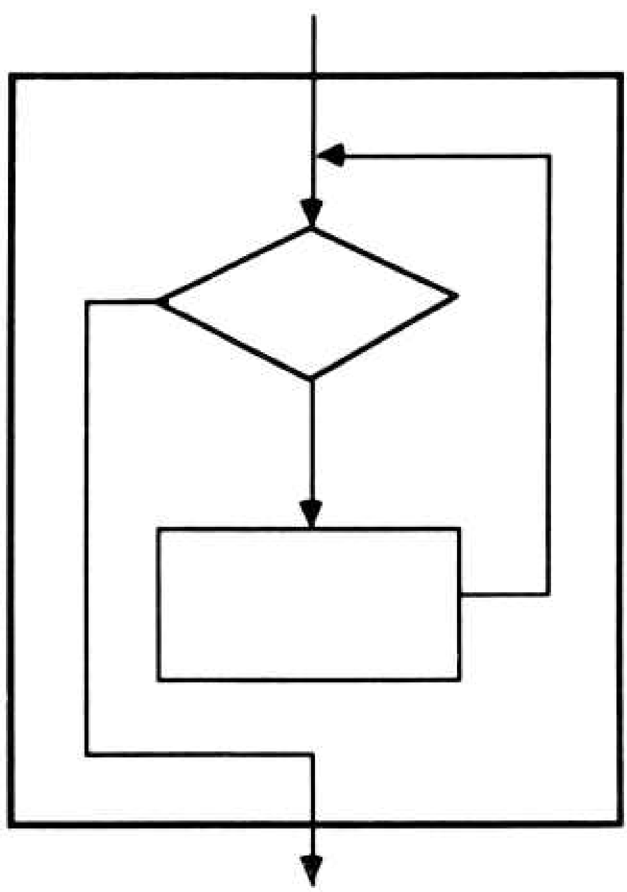
## While Statement Syntax

```
while_statement ::=
    WHILE condition LOOP
       sequence_of_statements
    END LOOP;
```

## Static Semantic Rule:

1. The Condition Must Be A Boolean Expression

## While Statement Flowchart



# FLAG CONTROLLED LOOP EXAMPLE

## Sum_Positive_With_Flag Procedure

```
WITH Text_IO;
PROCEDURE Sum_Positive_With_Flag IS
  Number: Integer;
  Sum:Integer := 0;
  Positive: Boolean := True;
  PACKAGE Int_IO IS NEW
    Text_IO.Integer_IO(Integer);
BEGIN
  WHILE Positive LOOP
    Int_IO.Get(Number);
    IF Number > 0 THEN
      Sum := Sum + Number;
      Positive := True;
    ELSE
      Positive := False;
    END IF;
  END LOOP;
  Text_IO.Put("The Sum Is ");
  Int_IO.Put(Sum);
  Text_IO.New_Line;
END Sum_Positive_With_Flag;
```

Important Point:

1. The Flag Controlling The Loop Must Be Initialized Prior To The Loop And Must Be Set Within The Loop

# SENTINEL CONTROLLED LOOP EXAMPLE

## Sum_Positive_With_Sentinel Procedure

```
WITH Text_IO;
PROCEDURE Sum_Positive_With_Sentinel IS
  Number: Integer;
  Sum:Integer := 0;
  PACKAGE Int_IO IS NEW
    Text_IO.Integer_IO(Integer);
BEGIN
  Int_IO.Get(Number);
  WHILE Number > 0 LOOP
    Sum := Sum + Number;
    Int_IO.Get(Number);
  END LOOP;
  Text_IO.Put("The Sum Is ");
  Int_IO.Put(Sum);
  Text_IO.New_Line;
END Sum_Positive_With_Sentinel;
```

## Important Points:

1. Controlling A Loop With A Sentinel Requires Reading The Input Twice
   a) A "Priming Read" Is Required Prior To The Loop
   b) Another Get Is Required At The End Of The Loop

2. The Sentinel Must Be Compared In The While Condition To Determine Loop Termination

# DEFINITE VS INDEFINITE ITERATION

Major Issue:

1. Every For Loop Can Be Replaced By A While Loop, The Reverse Is Not True

```
FOR Index IN Lower..Upper LOOP
  sequence_of_statements
END LOOP;

Index := Lower;
WHILE Index <= Upper LOOP
  sequence_of_statements
  Index := Index + 1;
END LOOP;
```

# INFINITE LOOPS

Nonterminating While Loop

```
Index := 1;
WHILE Index /= 0 LOOP
  Index := Index + 1;
END LOOP;
```

Important Point:

1. While Loops That Do Not Converge To The Termination Condition Are Infinite Loops

# LOOP AND EXIT STATEMENTS

## Loop Statement Syntax

loop_statement ::=

```
LOOP
   sequence_of_statements
END LOOP;
```

## Exit Statement Syntax

exit_statement ::=

```
EXIT [ WHEN condition ] ;
```

## Static Semantic Rule:

1. An Exit Statement Can Only Appear Within The Body Of A For, While Or Loop Statement

## Loop Exit Combined Syntax

```
LOOP
   sequence_of_statements
   EXIT [ WHEN condition ] ;
   sequence_of_statements
END LOOP;
```

## Important Points:

1. A Loop Statement Is Most Useful When The Loop Terminating Condition Can Not Be Determined At The Top Of The Loop

# LOOP STATEMENT EXAMPLE

## Count Letters Program

```
WITH Text_IO;
PROCEDURE Count_Letters IS
  Lower_Case, Upper_Case: Natural := 0;
  Char: Character;
  PACKAGE Count_IO IS NEW Text_IO.Integer_IO
    (Natural);
BEGIN
  Text_IO.Put_Line("Enter a sentence");
  WHILE NOT Text_IO.End_Of_Line LOOP
    Text_IO.Get(Char);
    EXIT WHEN Char = ' ';
    IF Char IN 'a'..'z' THEN
      Lower_Case := Lower_Case + 1;
    ELSIF Char IN 'A'..'Z' THEN
      Upper_Case := Upper_Case + 1;
    END IF;
  END LOOP;
  Text_IO.Put("Number of lower case = ");
  Count_IO.Put(Lower_Case);
  Text_IO.New_Line;
  Text_IO.Put("Number of upper case = ");
  Count_IO.Put(Upper_Case);
  Text_IO.New_Line;
END Count_Letters;
```

# PROCEDURES

## Procedure Concept

Procedures Differ From Functions In That They Do Not Return Values, But Their Parameters Can Return Values

## Procedure Declaration Syntax

```
procedure_declaration ::=
    PROCEDURE identifier [ formal_parameters ] IS
        { declarations }
    BEGIN
        sequence_of_statements
    END identifier;

formal_parameters ::=
    ( formal_parameter { , formal_parameter)

formal_parameter ::=
    identifier {, identifier} : [ mode ] type

mode ::=
    IN | OUT | IN OUT
```

## Static Semantic Rules:

1. The Type Of The Actual And Formal Parameters Must Match
2. Parameters Of In Mode Are Read Only And Parameters Of Out Mode Are Write Only

# PROCEDURE EXAMPLE

## Above And Below Procedure

```
WITH Text_IO;
PROCEDURE Above_And_Below(
  Number: IN Integer;
  Above_Count: OUT Natural;
  Below_Count: OUT Natural)
IS
  Value: Integer;
  Above, Below: Natural := 0;
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
BEGIN
  Int_IO.Get(Value);
  WHILE Value /= Number LOOP
    IF Value > Number THEN
      Above := Above + 1;
    ELSE
      Below := Below + 1;
    END IF;
    Int_IO.Get(Value);
  END LOOP;
  Above_Count := Above;
  Below_Count := Below;
END Above_And_Below;
```

## Important Point:

1. Out Parameters Are Write Only

# PROGRAM CORRECTNESS

Axiomatic Semantics

A Method For Defining The Meaning Of Procedures That Uses Assertions, Statements Of Formal Logic Or English,

Assertions

Precondition

An Assertion That Is True Prior To The Execution Of A Procedure

Postcondition

An Assertion That Is True After The Execution Of A Procedure

Loop Invariant

An Assertion That Is True Prior To The Execution Of A Loop, After Each Iteration And After The Execution Of The Loop Is Completed

Proof Of Correctness

The Precondition And Postcondition Define The Meaning Of A Procedure

A Proof Can Establish Correctness, That The Algorithm Accomplishes The Goal The Meaning Defines

Establishing That The Loop Terminates Is Necessary To Prove Total Correctness

# LOOP INVARIANT EXAMPLE

## Quotient Remainder Procedure

```
PROCEDURE Quotient_Remainder (
  Dividend: IN Integer;    -- d1
  Divisor: IN Integer;     -- d2
  Final_Quotient: OUT Integer;
  Final_Remainder: OUT Integer)
IS
  Quotient: Integer;  --q
  Remainder: Integer; --r
BEGIN
  --Precondition
  --(d1 ≥ 0) ∧ (d2 > 0)
  Quotient := 0;
  Remainder := Dividend;
  WHILE Remainder >= Divisor LOOP
    --Loop Invariant
    --(d1 = q * d2 + r) ∧ (r ≥ d2 > 0)
    Remainder := Remainder - Divisor;
    Quotient := Quotient + 1;
  END LOOP;
  --Postcondition
  --(d1 = q * d2 + r) ∧ (d2 > r ≥ 0)
  Final_Quotient := Quotient;
  Final_Remainder := Remainder;
END Quotient_Remainder;
```

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 8

# SCALAR DATA TYPES AND EXPRESSIONS

# SCALAR DATA TYPES

Ada Data Type Hierarchy

- Scalar Data Types
  - Discrete Types
    - Integer Types
    - Enumeration Types
      - Boolean Type
      - Character Type
  - Real Types
    - Floating Point
    - Fixed Point
- Composite Data Types
  - Array Types
  - Record Types
- Access Types

Terminology:

<u>Scalar Data Type</u>: A Data Type Whose Elements Consist Of A Single Value

<u>Discrete Type</u>: A Data Type Whose Elements Have Successors And Predecessors

Important Point:

1. Access Types Will Not Be Discussed In This Course

# NUMERIC OPERATORS

| Operation | Operator | Left | Right | Result |
|---|---|---|---|---|
| Addition | + | Numeric | Numeric | Numeric |
| Subtraction | - | Numeric | Numeric | Numeric |
| Multiplication | * | Numeric | Numeric | Numeric |
| Division | / | Numeric | Numeric | Numeric |
| Remainder | REM | Integer | Integer | Integer |
| Modulo | MOD | Integer | Integer | Integer |
| Exponentiation | ** | Integer | Integer | Integer |
| Exponentiation | ** | Float | Integer | Float |

Important Points:

1. The Remainder And Modulo Operators Are Only Defined On Integer Operands
2. The Remainder And Modulo Operators Produce The Same Results On Positive Integers
3. The Type Of The Left And Right Operands Must Be The Same Except In The Case Of Exponentiation

# EXPRESSION EVALUATION

## Expression Evaluation Rules

1. Parentheses

2. Precedence

| | |
|---|---|
| Highest | `ABS **` |
| Second Highest | `MOD REM * /` |
| Third Highest | `+ -` (Unary) |
| Lowest | `+ -` (Binary) |

3. Left To Right Associative

## Expression Evaluation Examples

`(2 + 3) * 5`  `25`

Parentheses Force Addition Before Multiplication

`8 - 6 / 2`  `5`

Precedence Causes Division Before Subtraction

`9 - 2 - 1`  `6`

Associativity Ensures Left Subtraction Before Right

## Expressions Requiring Parentheses

`2 ** 2 ** 3`

The Above Expression Is Syntactically Incorrect, Ada Requires That It Be Parenthesized

# TYPE CONVERSION

Major Issue:

Integer And Floating Point Number Can Be Mixed If Explicit Type Conversion Is Used

Type Conversion Fxample

```
I: Integer;
F: Float;
I := Integer(F); --Rounded
F := Float(I);
```

# FLOATING POINT NUMBERS

Floating Point Representation

Mantissa Controls Precision
Exponent Controls Range

Terminology:

Underflow: A Number Which Can Not Be Represented Because It Is Too Close To Zero

Overflow: A Number Which Can Not Be Represented Because It Is Too Far From Zero

Representation Error: Error Which Occurs Converting Decimal Numbers Such As 0.1 To Binary

# NUMERIC EXAMPLE

## Sphere Formulas

Volume = $\frac{4}{3}\pi r^3$ Surface Area = $4\pi r^2$

## Sphere Calculations Program

```
WITH Text_IO;
PROCEDURE Sphere_Calculations IS
  Pi: CONSTANT Float := 3.14159;
  Radius,Volume,Surface_Area: Float;
  PACKAGE Flt_IO IS NEW Text_IO.Float_IO
    (Float);
BEGIN
  Text_IO.Put("Enter Sphere Radius: ");
  Flt_IO.Get(Radius);
  Volume := (4.0/3.0) * Pi * Radius ** 3;
  Surface_Area := 4.0 * Pi * Radius ** 2;
  Text_IO.Put("Volume = ");
  Flt_IO.Put(Volume);
  Text_IO.New_Line;
  Text_IO.Put("Surface Area = ");
  Flt_IO.Put(Surface_Area);
  Text_IO.New_Line;
END Sphere_Calculations;
```

## Important Point:

1. The Types Around ** Do Not Match But Are Compatible With Function Specification O

# LOGICAL OPERATORS

| X | Y | NOT X | X AND Y | X OR Y | X XOR Y |
|---|---|---|---|---|---|
| True | True | False | True | True | False |
| True | False | False | False | True | True |
| False | True | True | False | True | True |
| False | False | True | False | False | False |

## Precedence Of All Operators

| | |
|---|---|
| Highest | NOT ABS ** |
| Second Highest | MOD REM * / |
| Third Highest | + - (Unary) |
| Fourth Highest | + - & (Binary) |
| Fifth Highest | < <= > >= = /= |
| Lowest | AND OR XOR |

## Compound Boolean Expressions

| | |
|---|---|
| 3 > 2 AND 8 = 4 | False |
| 2 <= 2 OR 6 = 5 | True |

## Important Point:

1. Compound Expressions With Mixed Logical Operators Require Parentheses

   X AND Y OR Z     Syntax Error

# SHORT CIRCUIT OPERATORS

Short Circuit Principle

```
False AND Anything = False

True OR Anything = True
```

Short Circuit Operators

Short Circuit Conjunction `AND THEN`

Short Circuit Disjunction `OR ELSE`

Short Circuit Evaluation

1. Left Operand Is Always Evaluated First

2. Right Operand Is Not Evaluated When
   a) Left Operand Is False And Operator Is And Then
   b) Right Operand Is True And Operator Is Or Else

Short Circuit Example

```
Y /= 0 AND THEN X / Y > 0
```

Use Of The Short Circuit Operator Prevents Evaluation Of The Right Operand When The Left Operator Is False, Which Prevents Division By Zero

Important Point:

1. With Ordinary Logical Operators, Either Operand May Be Evaluated First And Both Are Always Evaluated

# BOOLEAN EXPRESSION EXAMPLE

## Diagnose Program

```
WITH Text_IO;
PROCEDURE Diagnose IS
  Smiling, Laughing, Singing, Frowning,
    Weeping, Wailing: Boolean;
BEGIN
  Smiling:= False;
  Laughing := False;
  Singing := True;
  Frowning := False;
  Weeping := True;
  Wailing := True;
  IF Smiling AND THEN
    (Laughing OR Singing) THEN
    Text_IO.Put_Line("Must be happy");
  ELSIF Frowning OR ELSE
    (Weeping AND Wailing) THEN
    Text_IO.Put_Line
    ("Must be angry or depressed");
  END IF;
END Diagnose;
```

## Important Points:

1. The Parentheses In These Boolean Expressions Are Required

2. Non Short Circuit Operators Could Have Been Used

# CHARACTER DATA TYPE

## Nonprintable Characters

### Nonprintable Characters Can Be Represented By Name

| | |
|---|---|
| `ASCII.BEL` | Bell Character |
| `ASCII.CR` | Carriage Return |
| `ASCII.LF` | Line Feed |

## Control Characters Program

```
WITH Text_IO;
PROCEDURE Control_Characters IS
BEGIN
  Text_IO.Put_Line
    ("This one put_line statement" &
    ASCII.CR & ASCII.LF &
    "generates several lines of " &
    ASCII.CR & ASCII.LF &
    "text on the screen  " & ASCII.CR &
    "and overstrikes the third line " &
    "because no linefeed was issued");
  Text_IO.New_Line(3);
  Text_IO.Put_Line
    ("Non-printing characters can ring " &
    "the terminals bell" & ASCII.BEL);
  Text_IO.New_Line(3);
  Text_IO.Put_Line("ASCII.ESC is also " &
    "used for cursor-positioning");
END Control_Characters;
```

# CASE STATEMENT

Case Statement Syntax

case_statement ::=
    CASE expression IS
        case_statement_alternative
        {case_statement_alternative}
    END CASE;

case_statement_alternative ::=
    WHEN choice {| choice} =>
    sequence_of_statements

choice ::=simple_expression | discrete_range | OTHERS |
    *component*_simple_name

Semantic Rules:

1. The Expression Must Be Of A Discrete Type, Integer Or Enumerated
2. The Type Of The Choices Must Match The Type Of The Expression
3. Every Possible Value Of The Expression Must Be Specified By Exactly One Of The Choices Or By Others
4. Every Choice Must Be Static (Able To Be Evaluated At Compile Time)
5. If Used, The When Others Clause Must Be Last

# CASE STATEMENT EXAMPLE

## Favorite Colors And Numbers Program

```
WITH Text_IO;
PROCEDURE Favorite_Colors_And_Numbers IS
   TYPE Color_Type IS (Red, Puce, Blue,
     Purple, White, Magenta, Beige);
   Favorite_Number: Integer := 13;
   Favorite_Color: Color_Type := Beige;
BEGIN
  CASE Favorite_Number IS
    WHEN Integer'First..-1 =>
       Text_IO.Put_Line("How avant-garde!");
    WHEN 13 =>
      Text_IO.Put_Line("How bold");
    WHEN 0..12 | 14..99 =>
       Text_IO.Put_Line("How ordinary");
    WHEN OTHERS =>
      Text_IO.Put_Line("You think big");
  END CASE;
  CASE Favorite_Color IS
    WHEN Red | White | Blue =>
      Text_IO.Put_Line("How patriotic!");
    WHEN Puce | Purple | Magenta | Beige =>
       Text_IO.Put_Line("How unusual!");
   END CASE;
END Favorite_Colors_And_Numbers;
```

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 9

# SIMPLE ARRAY TYPES

# ARRAY DECLARATIONS

## Array Definition Syntax

array_definition ::= `ARRAY` (discrete_range) `OF` type

## Constrained Array Type Declarations:

```
-- An Array For Hours Worked For A Week

TYPE Days_Of_Week IS (Mon, Tue, Wed, Thu,
   Fri, Sat, Sun);

TYPE Hours IS ARRAY(Days_Of_Week) OF Float;

-- An Array To Contain 10 Logical Values

TYPE Boolean_Array IS ARRAY(1..10) OF
   Boolean;

-- An Array Of Grade Frequencies

SUBTYPE Grades IS INTEGER RANGE 0..100;

TYPE Frequencies IS ARRAY(Grades) OF
   Natural;
```

## Important Points:

1. Arrays Collect Together Data Elements Of The Same Type
2. The Type Of The Subscripts Must Be A Discrete Type, But The Components May Be Any Type

# ARRAY SUBSCRIPTS

## Role Of Array Subscripts

Array Subscripts Select One Element From The Array

```
Schedule: Hours;
```

| | |
|---|---|
| `Schedule(Mon)` | 8.0 |
| `Schedule(Tue)` | 8.0 |
| `Schedule(Wed)` | 8.0 |
| `Schedule(Thu)` | 8.0 |
| `Schedule(Fri)` | 8.0 |
| `Schedule(Sat)` | 0.0 |
| `Schedule(Sun)` | 0.0 |

## Important Point:

1. A Subscript Can Be A Constant Or A Variable

## Types Of Array Subscripts

Semantically Significant Subscripts

Subscripts Of Arrays That A Collection Of Values
(Subscripts Have No Particular Meaning)

# ARRAY AGGREGATES

Terminology:

<u>Array Aggregate</u>: A Literal Constant Which Represents The Value Of A Complete Array

Array Aggregate Syntax

aggregate ::=
(component_association {, component_association})

component_association ::=
[choice { | choice} =>] expression

choice ::= simple_expression | discrete_range | `OTHERS`

Array Object Declaration:

```
Worked: Hours;
```

Array Aggregates Assignments:

```
Worked := (4.0,4.0,8.0,8.0,8.0,0.0,0.0);

Worked := (4.0,4.0,8.0,8.0,8.0,OTHERS =>
  0.0);

Worked := (Sat|Sun => 0.0, Wed..Fri => 8.0,
   Mon..Tue => 4.0);
```

Important Point:

1. Named Aggregates Values Can Appear In Any Order

# ARRAY WITH SEMANTICALLY SIGNIFICANT SUBSCRIPTS

## Grade Frequency Program

```
WITH Text_IO;
PROCEDURE Grade_Frequency IS
  SUBTYPE Grades IS Integer RANGE 0..100;
  SUBTYPE Students IS Integer RANGE 1..30;
  TYPE Frequencies IS ARRAY(Grades) OF
    Natural;
  Grade: Grades;
  Frequency: Frequencies := (OTHERS => 0);
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
BEGIN
  FOR Student_Index IN Students LOOP
    Text_IO.Put("Enter Grade: ");
    Int_IO.Get(Grade);
    Frequency(Grade) := Frequency(Grade)
      + 1;
  END LOOP;
  FOR Grade_Index IN Grades LOOP
    Int_IO.Put(Grade_Index);
    Int_IO.Put(Frequency(Grade_Index));
    Text_IO.New_Line;
  END LOOP;
END Grade_Frequency;
```

# ARRAY AS COLLECTION

## Average Measurements Program

```
WITH Text_IO;
PROCEDURE Average_Measurements IS
  SUBTYPE Num_Measurements_Type IS Integer
    RANGE 1..10;
  TYPE Measurement_Array_Type IS
    ARRAY(Num_Measurements_Type) OF Float;
  Measurement_Array: Measurement_Array_Type;
  Total, Average: Float;
  PACKAGE Measurement_IO IS NEW Text_IO.
    Float_IO(Float);
BEGIN
  FOR I IN 1..10 LOOP
    Text_IO.Put_Line("Enter a measurement");
    Measurement_IO.Get
      (Measurement_Array(I));
  END LOOP;
  Total := 0.0;
  FOR I IN 1..10 LOOP
    Total := Total + Measurement_Array(I);
  END LOOP;
  Average := Total / 10.0 ;
  Text_IO.Put_Line("The average is ");
  Measurement_IO.Put(Average);
  Text_IO.New_Line;
END Average_Measurements;
```

# ARRAYS OF AN ANONYMOUS TYPE

Anonymous Array Declaration Syntax

> anonymous_array_declaration ::=
> identifier_list: `ARRAY` (discrete_range) `OF` type;

Anonymous Array Object Declarations:

```
Array_1,Array_2: ARRAY (1..10) OF Integer;

Array_3: ARRAY (1..10) OF Integer;
```

Important Points:

1. Anonymous Array Object Declarations Should Only Be Used For "One Of A Kind" Arrays
2. All Three Arrays Are All Of Different Types

# ARRAY PARAMETERS

Terminology:

> Call By Reference: The Address Of A Parameter Is Passed Instead Of The Parameter Itself

Important Point:

1. Array Parameters Can Be Passed By Reference Or By Value-Result, The Compiler Chooses The Method

# ARRAY PARAMETER EXAMPLE

## Sum Array Program

```
WITH Text_IO;
PROCEDURE Sum_Array IS
  TYPE Integer_Array_Type IS
    ARRAY(1..5) OF Integer;
  Actual_Array: Integer_Array_Type :=
    (21, 4, 5, 11, 26);
  Sum: Integer;
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
  PROCEDURE Calculate_Sum(Formal_Array: IN
    Integer_Array_Type;
    Sum: OUT Integer) IS
    Total: Integer;
  BEGIN
    Total := 0;
    FOR I IN 1..5 LOOP
      Total := Total + Formal_Array(I);
    END LOOP;
    Sum := Total;
  END Calculate_Sum;
BEGIN
  Calculate_Sum(Actual_Array, Sum);
  Text_IO.Put_Line("The sum is ");
  Int_IO.Put(Sum);
  Text_IO.New_Line;
END Sum_Array;
```

# STRING TYPES

## String Literals

A Sequence Of Characters Enclosed In Double Quotes
String Literals Must Not Cross Line Boundaries

## String Assignment

String Lengths Must Match

```
Some_String: String(1..5);
Some_String := "This String"; --Error
```

## String Concatenation

Creates A Single String From Two Strings

```
Text_IO.Put("This Output Is Too Long" &
  "To Fit One A Single Line");
```

## String Slices

Extracts Substrings From Strings

```
Any_String := "A Several Word String";
Text_IO.Put(Any_String(3..9));
  -- Outputs The Word "Several"
```

## String Comparisons

Left-Most Characters Are Compared First

```
"Cat" > "Dog"    False
"Car" < "Cart"   True
```

# STRING SLICE EXAMPLE

## Slice Program

```
WITH Text_IO;
PROCEDURE Slice IS
  SUBTYPE Bounds_1 IS Integer RANGE 1..5;
  SUBTYPE Bounds_2 IS Integer RANGE 1..11;
  Pattern: String(Bounds_1);
  Text: String(Bounds_2) := "Sample Text";
  Match: Boolean := False;
  Upper,Finish: Natural;
  PACKAGE Bool_IO IS NEW
    Text_IO.Enumeration_IO(Boolean);
BEGIN
  Text_IO.Put("Enter Pattern: ");
  Text_IO.Get(Pattern);
  Upper :=  Bounds_2'Last -
    Bounds_1'Last + 1;
  FOR Start IN Bounds_2'First..Upper LOOP
    Finish := Start + Bounds_1'Last - 1;
    IF Pattern = Text(Start..Finish) THEN
        Match := True;
        EXIT;
    END IF;
  END LOOP;
  Bool_IO.Put(Match);
END Slice;
```

# ARRAY SEARCHING

## Linear Search Procedure

```
WITH Text_IO;
PROCEDURE Linear_Search IS
  SUBTYPE Bounds IS Integer RANGE 1..10;
  Table: ARRAY(Bounds) OF Integer :=
    (31,15,8,34,5,81,2,97,30,95);
  Value: Integer;
  Found: Boolean;
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
BEGIN
  Text_IO.Put("Enter Value: ");
  Int_IO.Get(Value);
  FOR Index IN Bounds LOOP
    Found := Table(Index) = Value;
    EXIT WHEN Found;
  END LOOP;
  IF Found THEN
    Text_IO.Put("Value In Table");
  ELSE
    Text_IO.Put("Value Not In Table");
  END IF;
  Text_IO.New_Line;
END Linear_Search;
```

## Important Point:

1. The Execution Time Increases Linearly

# BIG O CONCEPT

Execution Time Function For Linear Search

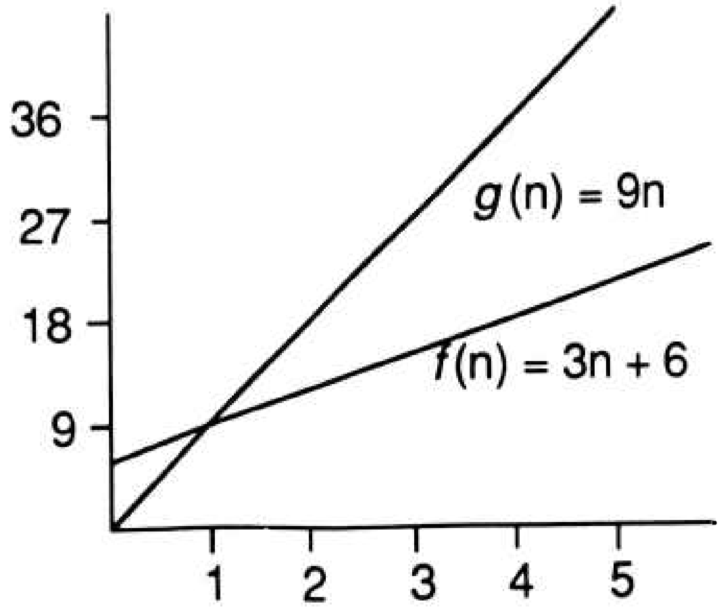
$f(n) = 3n + 6$, Where n Is The Table Size

Formal Definition Of Big-O:

$f(n) \in O(g(n))$ If And Only If There Exist Two Constants m and $n_0$ such that $|f(n)| \leq m|g(n)|$ for $n \geq n_0$

Big-O Proof

$3n + 6 \in O(n)$ Because $|3n + 6| \leq 9|n|$ for $n \geq 1$




Big-O Relationships

$O(\log x) \subseteq O(x) \subseteq O(x^2) \subseteq O(x^3) \subseteq O(e^x)$

$O(\log x)$ Is Most Efficient, $O(e^x)$ Is Least Efficient

# FUNCTION EXAMPLE

## Estimate Tax Procedure

```
WITH Text_IO;
PROCEDURE Estimate_Tax IS
  Income, Tax_Bill: Float;
  FUNCTION Federal_Taxes(Income,
    Tax_Rate: Float) RETURN Float IS
  BEGIN
    RETURN Income * Tax_Rate;
  END Federal_Taxes;
  PACKAGE Income_IO IS NEW Text_IO.Float_IO
    (Float);
BEGIN
  Text_IO.Put_Line("What is your income");
  Income_IO.Get(Income);
  IF Income <= 18000.00 THEN
    Tax_Bill := Federal_Taxes(Income, 0.15);
  ELSE
    Tax_Bill := Federal_Taxes
      (18000.00, 0.15);
    Tax_Bill := Tax_Bill + Federal_Taxes
      (Income - 18000.00, 0.31);
  END IF;
  Text_IO.Put("Your approximate taxes ");
  Income_IO.Put(Tax_Bill, 10, 2, 0 ) ;
  Text_IO.New_Line ;
END Estimate_Tax;
```

# FORMAL PARAMETERS, ACTUAL PARAMETERS AND LOCAL VARIABLES

Terminology

<u>Formal Parameters</u>: The Names Declared After The Function Name And Before The "Return" In A Function Declaration

<u>Actual Parameters</u>: The Expressions That Are Supplied In A Function Call

<u>Local Variables</u>: The Names Declared After The "Is" And Before The "Begin"

Static Semantic Rules:

1. The Type Of Corresponding Actual Parameters And Formal Parameters Must Match
2. Formal Parameters And Local Variables Can Only Be Accessed In The Function In Which They Are Defined
3. In Functions, Formal Parameters Act As Constants, They Are Read Only, They Cannot Be Assigned To

Parameter Association In Function Calls

Positional Association: Only The Actual Parameter Is Supplied

Named Association: Both The Actual And The Formal Parameters Are Supplied

# PACKAGES CONTAINING FUNCTIONS

## Package Concept

A Package Collects Together Functions Into A Library That Can Be Used By Many Programs

A Package Consists Of A Specification, Which Defines The Interface, And A Package Body, Which Contains The Implementation Details

## Package Specification Syntax

```
package_specification ::=
    PACKAGE identifier IS
        { function_specification }
    END identifier ;
```

## Function Specification Syntax

```
function_specification
    FUNCTION identifier [ formal_parameters ]
    RETURN type ;
```

## Package Body Syntax

```
package_body ::=
    PACKAGE BODY identifier IS
        { function_declaration }
    END identifier ;
```

## Important Point:

1. This Is A Very Simplified Package Syntax

# PACKAGE SPECIFICATION EXAMPLE

## Tax Package Specification

```
PACKAGE Tax_Package IS
  FUNCTION Federal_Taxes
    (Income, Tax_Rate: Float) RETURN Float;
  FUNCTION FICA_Taxes
    (Income: Float; Self_Employed: Boolean)
    RETURN Float;
-- FUNCTION State_Taxes
--   (Income: Float; State: String)
--   RETURN Float;
END Tax_Package;

-- Social security taxes only applied to the
-- first 54,000 of income
--
-- Medicare taxes not subject to an income
-- limit
--
-- Self_employed individuals pay both the
-- employee and employer share
--
-- A function specification is included as
-- a comment for state tax function
```

# PACKAGE BODY EXAMPLE

## Tax Package Body

```
PACKAGE BODY Tax_Package IS
  FUNCTION Federal_Taxes
    (Income, Tax_Rate: Float)
  RETURN Float IS
  BEGIN
    RETURN Income * Tax_Rate;
  END Federal_Taxes;
  FUNCTION FICA_Taxes
    (Income: Float; Self_Employed: Boolean)
  RETURN Float IS
    Rate, Tax_Bill: Float;
  BEGIN
    IF Self_Employed THEN
      Rate := 0.145;
    ELSE
      Rate := 0.0725;
    END IF;
    IF Income <= 54000.00 THEN
      Tax_Bill := Income * Rate;
    ELSE
      Tax_Bill := Income * Rate;
      Tax_Bill := Tax_Bill +
       (Income - 54000.00) * 0.0125;
    END IF ;
    RETURN Tax_Bill;
  END FICA_Taxes;
END Tax_Package;
```

# FUNCTION SEMANTICS

Denotational Semantics

A Method For Defining The Meaning Of Functions That Uses Mathematical Functions

Ada Functions As Mathematical Functions

The Domain Of The Mathematical Function Is The Cartesian Product Of The Types Of The Formal Parameters

The Co-domain Is The Type Returned By The Function

Function Example

```
FUNCTION Hypotenuse
  Side_1: Float;
  Side_2: Float)
RETURN Float;
```

$h\colon \mathrm{R} \times \mathrm{R} \rightarrow \mathrm{R}$

Mathematical Function Definition

The Mathematical Definition Of An Ada Function Should Depend Only On The Formal Parameters, The Local Variables Should Not Be A Part Of The Definition

Important Point:

1. A Function Specification Should Include The Ada Function Specification And A Comment Defining Its Meaning, Mathematics Or English Can Be Used

# EQUIVALENT FUNCTIONS

## Two Equivalent Functions

```
WITH Math_Lib;
FUNCTION Hypotenuse (Side_1,Side_2: Float)
RETURN Float IS
  Side_1_Squared,Side_2_Squared: Float;
  PACKAGE Math IS NEW Math_Lib(Float);
BEGIN
  Side_1_Squared := Side_1 ** 2;
  Side_2_Squared := Side_2 ** 2;
  RETURN Math.Sqrt(Side_1_Squared +
    Side_2_Squared);
END Hypotenuse;

WITH Math_Lib;
FUNCTION Hypotenuse (Side_1,Side_2: Float)
RETURN Float IS
  Sum_Of_Squares,Side_3: Float;
  PACKAGE Math IS NEW Math_Lib(Float);
BEGIN
  Sum_Of_Squares := Side_1 ** 2 +
    Side_2 **2;
  Side_3  := Math.Sqrt(Sum_Of_Squares);
  RETURN Side_3;
END Hypotenuse;
```

## Mathematical Definition

$h\colon \mathrm{R} \times \mathrm{R} \rightarrow \mathrm{R} \qquad h(s_1,s_2) = \sqrt{s_1{}^2 + s_2{}^2}$

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 6

# DEFINITE ITERATION

# DEFINITE ITERATION CONCEPT

Major Issues:

1. Programs Containing Only Conditional Control Have Forward Control Flow

2. Certain Problems Require Iterative Control To Be Solved

Terminology:

<u>Definite Iteration</u>: Control Mechanism That Permits The Repetition Of A Group Of Statements A Fixed (At Run-Time) Number Of Times

Important Points:

1. If The Number Of Repetitions Is Known At Compile-Time, Iteration Can Be Avoided By Repeating The Code At Fixed Number Of Times

2. If The Number Of Repetitions Is Not Known Until Run-Time, Definite Iteration Is Required

A Simple Problem Requiring Definite Iteration
Computing Triangular Numbers

1. Read In An Integer n, From The Keyboard

2. Compute The Sum Of The First n Integers

$$\sum_{i=1}^{n} i$$

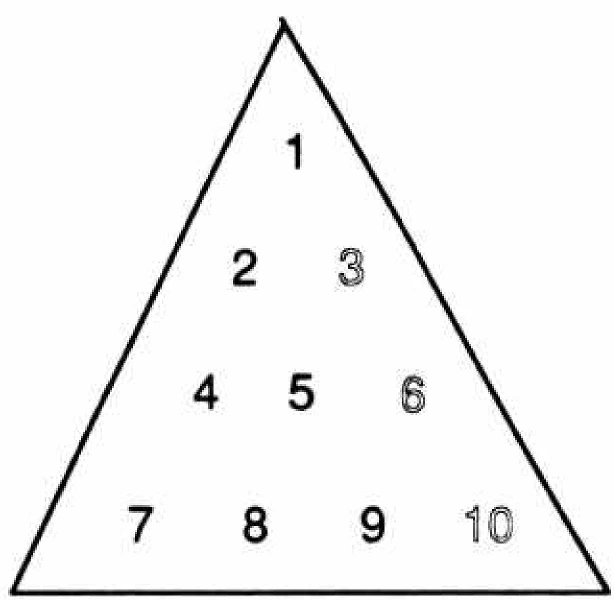
3. Print The nth Triangular Number, The Sum

# DEFINITE ITERATION EXAMPLE

## Triangular Numbers Program

```
WITH Text_IO;
PROCEDURE Triangular_Numbers IS
  Number, Triangle: Integer;
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
BEGIN
  Text_IO.Put("Enter Number: ");
  Int_IO.Get(Number);
  Triangle := 0;
  FOR Index IN 1..Number LOOP
    Triangle := Triangle + Index;
  END LOOP;
  Text_IO.Put("Triangular Number = ");
  Int_IO.Put(Triangle);
END Triangular_Numbers;
```

## Triangular Numbers



# FOR STATEMENT

For Statement Syntax

```
for_statement ::=
    FOR identifier IN [ REVERSE ] discrete_range LOOP
      sequence_of_statements
    END LOOP;
```

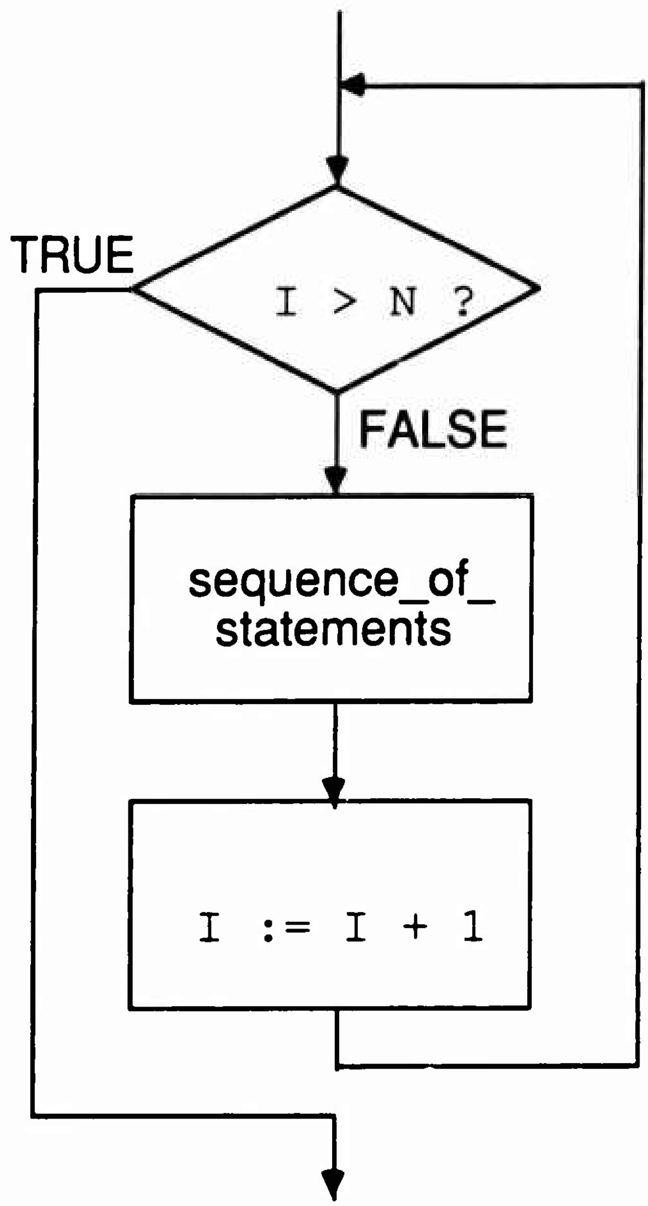
Static Semantic Rules:

1. The Loop Control Variable Is Implicitly Declared
   a) The Loop Control Variable Should Not Be Explicitly Declared, Doing So Creates Two Variables
   b) It Can Only Be Referenced From The Sequence Of Statements That Comprise The Body Of The Loop
   c) It Only Exists During The Execution Of The Loop, It Is Deallocated Afterward
2. The Type Of The Loop Control Variable, Defined By Its Range Of Values Must Be Discrete, Integer Or Enumerated
3. Within The Body Of The Loop, The Loop Control Variable Can Not Be Modified By Assignments
4. The Bounds Of The Loop Are Evaluated Once And Cannot Be Altered In The Body Of The Loop

# FOR STATEMENT FLOWCHART

## Ascending For Statement

```
FOR I IN 1..N LOOP
   sequence_of_statements
END LOOP;
```



# IF STATEMENT WITH ELSIF CLAUSES

## Categorize Courses Program

```
WITH Text_IO;
PROCEDURE Categorize_Courses IS
  TYPE Categories IS (Lower_Level,
    Upper_Level, Graduate, Invalid);
  Category: Categories;
  Course_Number: Integer;
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
  PACKAGE Categories_IO IS NEW Text_IO.
    Enumeration_IO(Categories);
BEGIN
  Text_IO.Put("Enter Course: ");
  Int_IO.Get(Course_Number);
  IF Course_Number < 100 THEN
    Category := Invalid;
  ELSIF Course_Number < 300 THEN
    Category := Lower_Level;
  ELSIF Course_Number < 500 THEN
    Category := Upper_Level;
  ELSE
    Category := Graduate;
  END IF;
  Text_IO.Put("Category Is ");
  Categories_IO.Put(Category);
  Text_IO.New_Line;
END Categorize_Courses;
```

# FORMAL SYNTAX

Backus-Naur Form (BNF)

BNF Is A Meta-Language For Describing The Syntax Of Programming Languages, It Is Both Precise And Concise

BNF Meta-Symbols

- ::= Is Defined By
- [] Optional (Zero Or One)
- {} Repetition (Zero Or More)
- | Choice (One Or The Other)

If Statement Syntax

```
if_statement ::=
    IF condition THEN
        sequence_of_statements
    { ELSIF condition THEN
        sequence_of_statements }
    [ ELSE
        sequence_of_statements ]
    END IF;
```

Historical Note:

BNF Was First Used To Describe The Syntax Of The Programming Language Algol-60

# STATIC SEMANTICS

## Static Semantics In English

| | Syntax | Static Semantics |
|---|---|---|
| Jump the in house of. | × | × |
| The car eats the apple. | √ | × |
| The boy hits the ball. | √ | √ |

## Static Semantic Rules

Static Semantic Rules Are Rules That Determine Whether A Statement Is Meaningful

## Syntax Rules With Semantic Information

condition ::=
*boolean*_expression

The Ada Language Reference Manual Italicizes Static Semantic Information In Its Syntax Rules

## Another Static Semantic Rule

Requiring That The Types Of The Two Operands Within A Condition Be The Same Is A Static Semantic Rule, These Rules Often Involve Type Information Or Type Checking

# DYNAMIC SEMANTICS

Dynamic Semantic Models

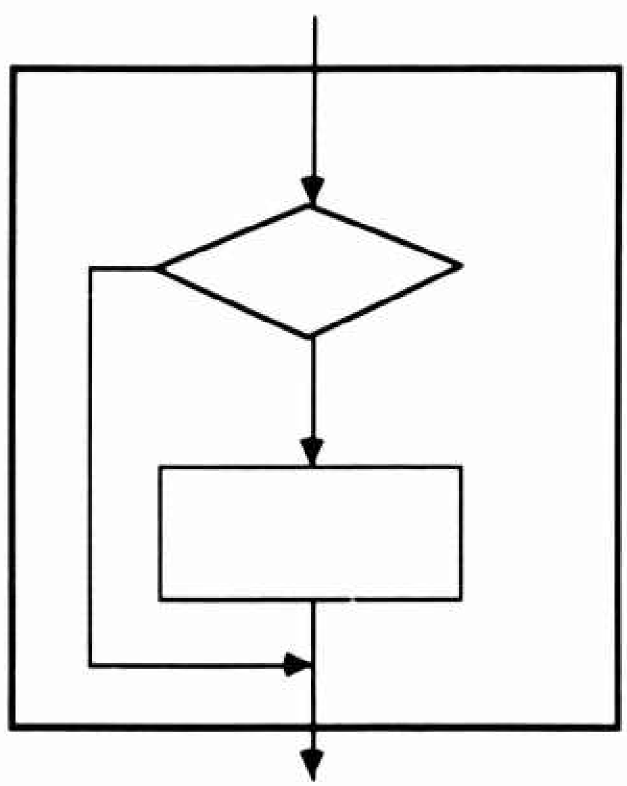
Dynamic Semantics Models Are Methods For Defining The Meaning Of Statements In Programming Languages

Operational Semantics

Operational Semantics Is One Model For Defining The Meaning Of Statements That Translates The Statement Into An Less Abstract Language, Unstructured Language

A Flowchart Is The Simplest Means Of Conveying The Operational Semantic Meaning Of A Statement

Simple If Statement Flowchart



# NESTED IF STATEMENTS

## Summer Day Program

```
WITH Text_IO;
PROCEDURE Summer_Day IS
  TYPE Months IS (Jan, Feb, Mar, Apr, May,
    Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  Month: Months;
  Day: Integer;
  PACKAGE Months_IO IS NEW Text_IO.
    Enumeration_IO(Months);
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
BEGIN
  Text_IO.Put("Enter Month and Day: ");
  Months_IO.Get(Month);
  Int_IO.Get(Day);
  IF Month = Jun THEN
    IF Day >= 21 THEN
      Text_IO.Put_Line("Summer");
    END IF;
  ELSIF Month IN Jul..Aug THEN
    Text_IO.Put_Line("Summer");
  ELSIF Month = Sep THEN
    IF Day < 21 THEN
      Text_IO.Put_Line("Summer");
    END IF;
  END IF;
END Summer_Day;
```

## Tracing Program Execution

Tracing The Execution Of A Program Can Be A Useful Technique For Understanding The Action Of A Program

## Trace Of Absolute Values Program

| | Number | Number < 0 |
|---|---|---|
| `Int_IO.Get(Number);` | -5 | |
| `IF Number < 0 THEN` | | True |
| `  Number = -Number` | 5 | |
| `END IF;` | | |

## Counting Paths

Two If Else Statements In Sequence Creates A Program With Four Possible Paths

An If Else Statement With One If Else Statement Nested In The If Part And One If Else Statement Nested In The Else Part Creates A Program With Four Paths

Summer Day Program Has 6 Paths

- Two Paths For June
- One Path For July And August
- Two Paths For September
- One Path For Other Months

Developing Testing Strategies

All Statements Strategy

Choose Test Data That Will Ensure That Each Statement In The Program Is Executed

All Paths Strategy

Choose Test Data That Will Ensure That Every Path Of The Program Is Executed

Test Data To Test Each Path Of Summer Day Program

| Condition | Test Data | Output |
|---|---|---|
| Month = Jun<br>Day < 21 | Jun 5 | None |
| Month = Jun<br>Day < 21 | Jun 30 | Summer |
| Month IN Jul..Aug | Jul 10 | Summer |
| Month = Sep<br>Day < 21 | Sep 8 | Summer |
| Month = Sep<br>Day >= 21 | Sep 25 | None |
| Month NOT IN<br>Jun..Sep | Feb 5 | None |

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 5

# FUNCTIONS

# DECLARING AND CALLING FUNCTIONS

## Function Concept

User Defined Functions Provide A First Capability For Procedural Abstraction, Defining A Common Function Once With The Ability To Call It Any Number Of Times

## Function Declaration Syntax

```
function_declaration ::=
    FUNCTION identifier [ formal_parameters ]
    RETURN type IS
      { declarations }
    BEGIN
      sequence_of_statements
    END identifier;

formal_parameters ::=
    ( formal_parameter { , formal_parameter } )

formal_parameter ::=
    identifier {, identifier} : type
```

## Function Call Syntax

```
function_call ::=
    identifier ( actual_parameter { , actual_parameter } )

actual_parameter ::=
    [ identifier => ] expression
```

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 10

# SIMPLE RECORD TYPES

# RECORD DECLARATIONS

Syntax Rules:

record_type_definition ::=
```
RECORD
```
component_list
```
END RECORD
```

component_list ::= component_declaration
{component_declaration} | `NULL`

component_declaration ::= identifier_list : type_mark
[constraint] [ : = expression];

Record Type Declaration:

```
SUBTYPE Hours IS INTEGER RANGE 0..23;

SUBTYPE Minutes IS INTEGER RANGE 0..59;

TYPE Times IS
  RECORD
    Hour: Hours;
    Minute: Minutes;
  END RECORD;
```

Important Points:

1. Records Can Collect Together Data Elements Of Different Types
2. Variables Of Record Types With Initialized Components Are Initialized To Those Values

# RECORD AGGREGATES

Terminology:

Record Aggregate: A Literal Constant Which Represents The Value Of A Complete Record

Record Object Declaration:

```
Time: Times;
```

Positional Aggregate:

```
Time := (6,45); -- Is Equivalent To
Time.Hour := 6;
Time.Minute := 45;
```

Named Aggregate:

```
Time := (Minute => 45, Hour => 6);
```

# RECORD OPERATIONS

Component Selection

Assignment: Types Must Match

Relational Operators: = /= Predefined

Arithmetic And Logical Operators: None Predefined

# RECORD EXAMPLE

## Students Type Definition

```
SUBTYPE Ages IS Integer RANGE 0..120;
SUBTYPE SSNs IS Integer RANGE
  100_000_000..999_999_999 ;
TYPE Students IS
  RECORD
    Name: String(1..30);
    Age: Ages;
    SSN: SSNs;
  END RECORD;
```

## Get Procedure

```
WITH Text_IO;
PROCEDURE Get(Student: OUT Students) IS
  Last: Natural;
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
BEGIN
  Text_IO.Put_Line("Enter name");
  Student.Name := (OTHERS => ' ');
  Text_IO.Get_Line(Student.Name, Last);
  Text_IO.Put_Line("Enter student's age");
  Int_IO.Get(Student.Age);
  Text_IO.Put_Line("Enter student's SSN");
  Int_IO.Get(Student.SSN);
END Get;
```

# ARRAYS OF RECORD EXAMPLE

**Array Of Record Type Definition:**

```
TYPE Departments IS (CMSC,CMIS,IFSM);
SUBTYPE Course_Numbers IS INTEGER RANGE
  100..499;
SUBTYPE Enrollments IS INTEGER RANGE 0..45;
TYPE Courses IS
  RECORD
    Department: Departments;
    Course_Number: Course_Numbers;
    Enrollment: Enrollments;
  END RECORD;
TYPE Course_Lists IS ARRAY(1..25) OF
  Courses;
List: Course_Lists;

List(1).Course_Number := 130;
```

| | Department | Course_Number | Enrollment |
|---|---|---|---|
| List(1) | CMSC | 130 | 42 |
| List(2) | CMSC | 135 | 35 |
| . . . | | | |
| List(25) | CMSC | 430 | 21 |

# NESTED RECORDS

## Preliminary Type Definitions:

```
Hours_Per_Day: CONSTANT := 24;
Minutes_Per_Hour: CONSTANT := 60;
SUBTYPE Hours IS INTEGER RANGE
  0..Hours_Per_Day - 1;
SUBTYPE Minutes IS INTEGER RANGE
  0..Minutes_Per_Hour - 1;
TYPE Departments IS (CMSC,CMIS,IFSM);
SUBTYPE Course_Numbers IS INTEGER RANGE
  100..499;
SUBTYPE Sections IS INTEGER RANGE
  1000..9999;
TYPE Days IS (Monday, Tuesday, Wednesday,
  Thursday, Friday, Saturday, Sunday);
```

## Nested Record Type Definition

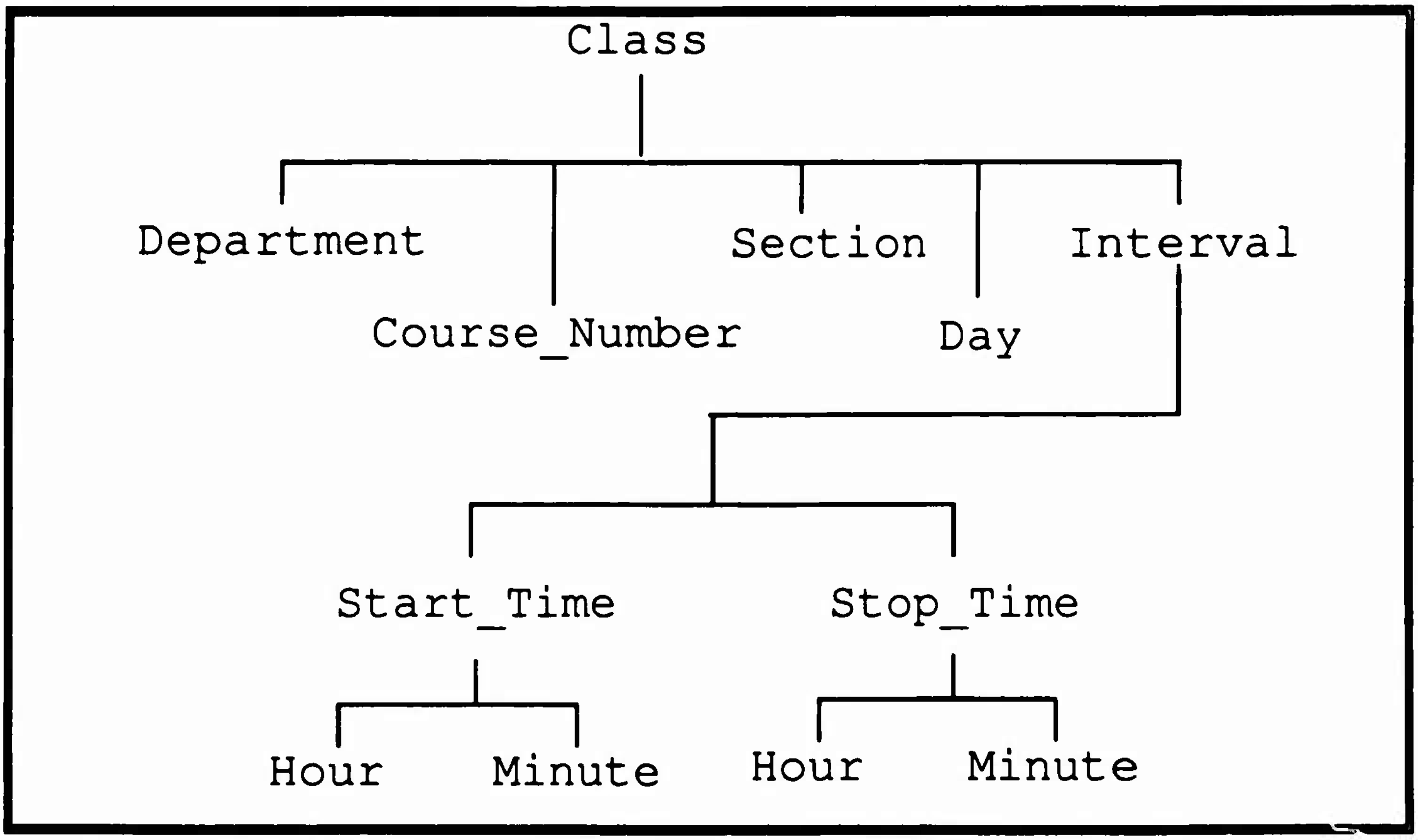
```
TYPE Times IS
  RECORD
    Hour: Hours;
    Minute: Minutes;
  END RECORD;
TYPE Intervals IS
  RECORD
    Start_Time: Times;
    Stop_Time: Times;
  END RECORD;
```

# DOUBLY NESTED RECORDS

**Doubly Nested Record Type Definition:**

```
TYPE Classes IS
  RECORD
    Department: Departments;
    Course_Number: Course_Numbers;
    Section: Sections;
    Day: Days;
    Interval: Intervals;
  END RECORD;
Class: Classes;

Class.Interval.Start_Time.Hour := 7;
```

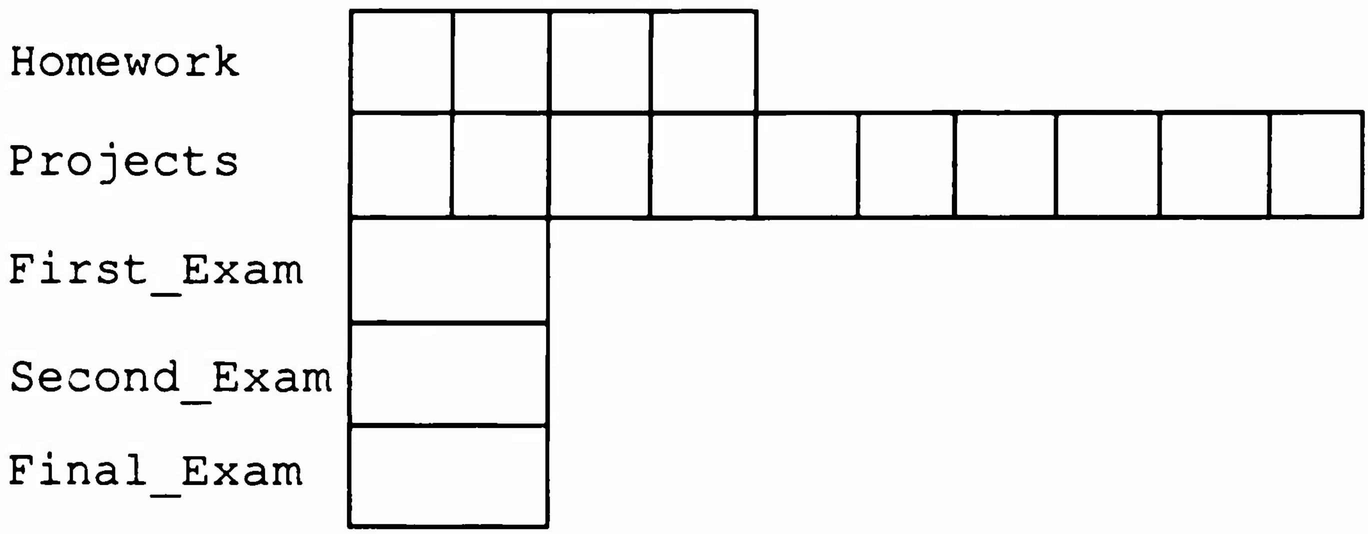


# RECORDS CONTAINING ARRAYS

## Record Containing Array Type Definition:

```
SUBTYPE Percents IS INTEGER RANGE 0..100;
TYPE Homeworks IS ARRAY(1..10) OF Percents;
TYPE Projects IS ARRAY(1..4) OF Percents;
TYPE Grades IS
  RECORD
    Homework: Homeworks;
    Project: Projects;
    First_Exam, Second_Exam, Final:
      Percents;
  END RECORD;
Grade: Grades;

Grade.Homework(2) := 100;
```



# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 11

# PROCEDURAL ABSTRACTION

# PHASES OF SOFTWARE DEVELOPMENT

Requirements

- Written Document Specifying Systems Requirements

Analysis And Design

- Analysis Of Requirements To Determine Hardware And Software Design
- System Design By Decomposition Into Subsystems Including Interface Specifications

Coding And Unit Testing

- Translation Of Design Of Individual Units Into A Specific Programming Language
- Testing Of Individual Units Using Drivers Or Stubs

Integration And Testing

- Integration Of Individual Units To Uncover Interface Problems
- Testing System Against Original Requirements Specification

Operation And Maintenance

- Operation Of System
- Maintenance Includes Fixing Problems As Discovered And Adding Necessary Enhancements

# TOP-DOWN DESIGN EXAMPLE

Problem: Design A Spreadsheet Program

Major Steps

1. Perform Initial Program Setup
2. Get A Command From The User
3. While The Command Is Not QUIT

   LOOP

   4. Execute The Command
   5. Display The Results Of The Command
   6. Get A Command From The User

   END LOOP

7. Quit

Important Points:

1. The Design Should Include High-Level Algorithms And Initial Ideas For Data Structures
2. The Outlines Of A Main Procedure And The Required Subprograms Should Begin To Emerge
3. One Approach Is To Create Compilable Ada Modules With All Of The High-Level Concepts Written As Comments

# TOP-DOWN DESIGN REFINEMENTS

1. Initial Program Setup

Allocate The Minimum Amount Of Memory Needed

Clear The Screen, Print The Logo And Display The Main Menu

2. Get A Command

```
IF Valid Menu Choice Selected THEN
    IF The Menu Choice Has Sub-Choices THEN
        Display The Sub-Choices
    ELSE
        Get  Required Data
        Add Data To Command Record
    END IF
ELSIF The Right Arrow THEN
    IF Cursor Is On Rightmost Menu Item THEN
        Place Cursor On The Leftmost Menu Item
    ELSE
        Move Cursor One Menu Item To The Right
    END IF
ELSIF The Left Arrow THEN
    IF Cursor Is On Leftmost Menu Item THEN
        Place Cursor On The Rightmost Menu Item
    ELSE
        Move Cursor One Menu Item To The Left
    END IF
END IF
```

# UNIT TESTING STRATEGIES

Top-Down Testing

Testing Strategy

A Method Of Unit Testing For High-Level Subprograms Using Incomplete Lower Level Subprograms Called Stubs

Stubs

Stubs Typically Output A Message Indicating That They Have Been Called

Bottom-Up Testing

Testing Strategy

A Method Of Unit Testing For Low-Level Subprograms Using Skeletal Higher Level Subprograms Called Drivers

Drivers

Drivers Typically Call The Subprogram To Be Tested And Output The Result

Important Points:

1. Ada Subunits Are A Language Feature That Can Facilitate Unit Testing

2. The Syntax Of Subunits Will Be Studied In The Subsequent Course

# TOP-DOWN TESTING

## Data Processor Procedure With Stubs

```
WITH Text_IO;
PROCEDURE Data_Processor IS
  --
  PROCEDURE Read_Input IS
  BEGIN
    Text_IO.Put("Input Read");
    Text_IO.New_Line;
  END Read_Input;
  PROCEDURE Perform_Processing IS
  BEGIN
    Text_IO.Put("Processing Performed");
    Text_IO.New_Line;
  END Perform_Processing;
  PROCEDURE Write_Output IS
  BEGIN
    Text_IO.Put("Output Written");
    Text_IO.New_Line;
  END Write_Output;
  --
BEGIN
  Read_Input;
  Perform_Processing;
  Write_Output;
END Data_Processor;
```

# BOTTOM-UP TESTING

## Driver Procedure To Test Min Function

```
WITH Text_IO;
PROCEDURE Driver IS
  Value1, Value2, Minimum: Integer;
  PACKAGE Int_IO IS NEW Text_IO.Integer_IO
    (Integer);
  --
  FUNCTION Min(Left,Right: Integer)
  RETURN Integer IS
  BEGIN
    IF Left < Right THEN
      RETURN Left;
    ELSE
      RETURN Right;
    END IF;
  END Min;
  --
BEGIN
  Text_IO.Put("Enter Two Integers: ");
  Int_IO.Get(Value1);
  Int_IO.Get(Value2);
  Minimum := Min(Value1,Value2);
  Text_IO.Put("Minimum Is ");
  Int_IO.Put(Minimum);
  Text_IO.New_Line;
END Driver;
```

# BLOCK STATEMENT

Block Statement Syntax

```
block_statement ::=
    [block_simple_name:]
    [DECLARE
       declarative_part]
    BEGIN
       sequence_of_statements
    END [block_simple_name];
```

Semantic Rule:

1. Block Names Must Match If They Are Present

Important Points:

1. Blocks Localize Variables And Can Save Memory, But Small Subprograms Can Do The Same
2. Blocks Are Most Useful For Exceptions

Terminology:

<u>Frame</u>: A Block, A Procedure Or A Function

| | | |
|---|---|---|
| DECLARE<br>    declarations<br>BEGIN<br>    statements<br>END | PROCEDURE<br>    declarations<br>BEGIN<br>    statements<br>END | FUNCTION<br>    declarations<br>BEGIN<br>    statements<br>END |

# SCOPE AND VISIBILITY

Terminology:

<u>Scope</u>: Portion Of A Program For Which A Specific Identifier Is Defined

<u>Visibility</u>: Identifiers That Can Be Accessed From A Specific Point In A Program

Scope And Visibility Rules

1. The Scope Of Identifiers Is From Their Declaration Until The End Of The Frame In Which They Are Declared
2. When The Same Identifier Is Redeclared In An Inner Frame, Only The Inner Occurrence Is Directly Visible (Most Closely Nested Rule)
3. When The Same Identifier Is Redeclared In An Inner Frame, The Outer Occurrence Is Visible By Selection Using The Expanded Name
4. The Scope Of Formal Parameters Is The Same As The Scope Of Their Subprogram Name, Outside The Subprogram They Are Visible Only Within Named Parameter Associations

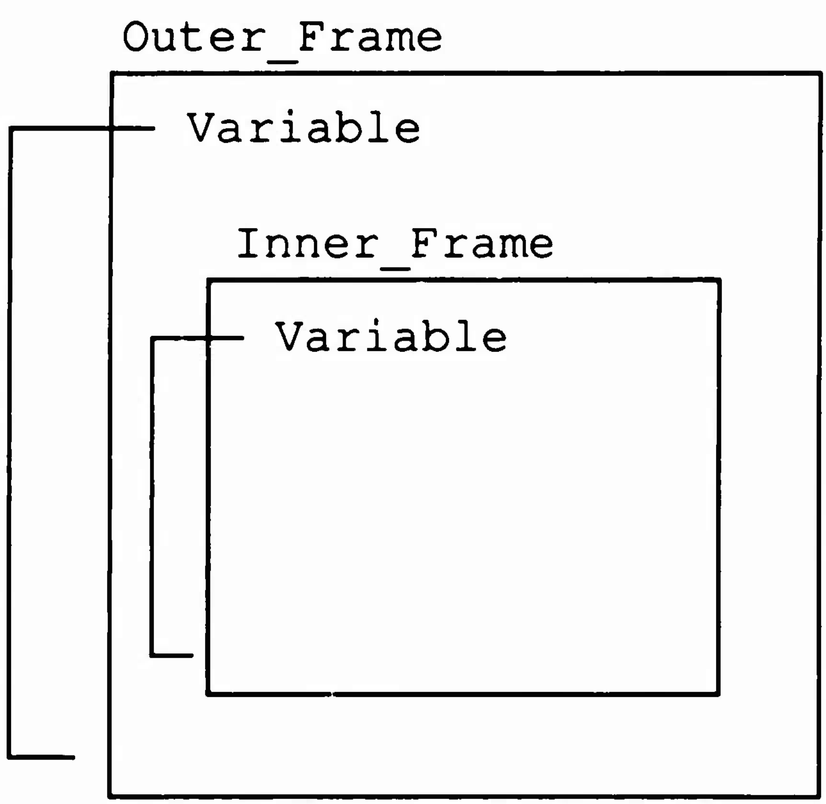
Accessing Hidden Identifiers

Hidden Identifiers Are Visible By Selection And Can Be Accessed Using An Expanded Name

# VISIBILITY BY SELECTION

## Expanded Name Example

```
PROCEDURE Outer_Frame IS
  Variable: Integer;
BEGIN
  Inner_Frame:
  DECLARE
    Variable: Character;
  BEGIN
    Inner_Frame.Variable := 'A';
    Outer_Frame.Variable := 1;
    Variable := 'B';
  END Inner_Frame;
END Outer_Frame;
```



# FILE INPUT/OUTPUT

Major Issue:

1. To Write Programs With More Than One Input File Or More Than One Output File, Explicit File Input/Output Is Required

Explicit File Processing

All Explicit Files Must Be Declared, Explicitly Opened And Explicitly Closed, All Input/Output Must Explicitly Name Which File Is Being Referenced

Text_IO Package

```
PACKAGE Text_IO IS
  TYPE File_Type IS LIMITED PRIVATE;
  TYPE File_Mode IS (In_File,Out_File);
  PROCEDURE Open(File: IN OUT File_Type;
    Mode: IN File_Mode; Name IN String;
    Form: IN String := "");
  PROCEDURE Close(File: IN OUT File_Type);
  FUNCTION End_Of_Line(File: IN File_Type)
    RETURN Boolean;
  FUNCTION End_Of_File(File: IN File_Type)
    RETURN Boolean;
  ...
  -- Many Other Subprograms
  ...
END Text_IO;
```

# EXPLICIT FILE EXAMPLE

## Copy File Procedure

```
WITH Text_IO; USE Text_IO;
PROCEDURE Copy_File IS
  Input_File, Output_File: File_Type;
  Char: Character;
BEGIN
  Open(Input_File,In_File,"IN.TXT");
  Create(Output_File,Out_File,"OUT.TXT");
  WHILE NOT End_Of_File(Input_File) LOOP
    WHILE NOT End_Of_Line(Input_File) LOOP
      Get(Input_File,Char);
      Put(Output_File,Char);
    END LOOP;
    Skip_Line(Input_File);
    New_Line(Output_File);
  END LOOP;
  Close(Input_File);
  Close(Output_File);
END Copy_File;
```

## Important Points:

1. This Program Copies The Line And File Structure Of The Input File To The Output File
2. The Output File Is Created Rather Than Opened Because It Does Not Already Exist

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 12

# DATA ABSTRACTION

# ABSTRACT DATA TYPES

Terminology:

Abstract Data Type: An Axiomatic Definition Of A Data Type Which Defines The Behavior Of The Data Type And Its Associated Operations

The Natural Numbers - **N**

Operations

Zero $\mathbf{0} : \varnothing \to \mathbf{N}$

Successor $\sigma : \mathbf{N} \to \mathbf{N}$

Peano Axioms

1. $\forall\, n,m \in \mathbf{N}: \sigma(n) = \sigma(m) \Rightarrow n = m$
2. $\forall\, n \in \mathbf{N}: \sigma(n) \neq \mathbf{0}$
3. $\forall\, M \ni M \subseteq \mathbf{N}: (\mathbf{0} \in M, n \in M \Rightarrow \sigma(n) \in M) \Rightarrow M = \mathbf{N}$

   Mathematical Induction Axiom

Supplemental Operation

Addition $+ : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$

Addition Axioms

1. $\forall\, n \in \mathbf{N}: n + \mathbf{0} = n$
2. $\forall\, n,m \in \mathbf{N}: n + \sigma(m) = \sigma(n + m)$

# DATA TYPE REPRESENTATIONS AND IMPLEMENTATIONS

Terminology:

<u>Data Type Representation</u>: A Way To Represent The Elements Of A Data Type

<u>Data Type Implementation</u>: A Way To Implement The Operations Of A Data Type Which Is Consistent With Its Abstract Definition

Natural Numbers Representations

Literal Representations

1. Roman Numerals
2. Arabic Numerals In Base 10
3. Arabic Numerals In Base 5

Internal Machine Representations

1. Natural
2 Binary Coded Decimal
3 ASCII

Natural Numbers Implementation

To Uncover The Details Of The Implementation Of The Arithmetic Operations For Natural Numbers, One Must Look Into The Arithmetic Logic Unit Of The CPU

# EXTERNAL VIEW AND INTERNAL DETAILS

External View

The Abstract Data Type Is The External View, It Describes The Operations Of The Data Type And Their Behavior

In Practice The Behavior Of A Data Type Might Best Be Described With English, Not Formal Mathematics

Major Issues:

1. The Exact Details Of How The Data Is Represented And The Operations Are Implemented Is Unimportant To The User, These Details Should Be Hidden

2. It Is Important That The Implementation Be Consistent With The Axioms, 2 + 2 Must Equal 4

Internal Details

Internal Details Can Be Determined By The Hardware, System Software Or Application Software

Important Points:

1. The Internal Machine Representation And Implementation For Fundamental Data Types Is Made By The Hardware Designers

2. The Representation And Implementation For User-Defined Data Types Must Be Made By The Software Designer

# ADA AND ABSTRACT DATA TYPES

Major Issue:

Ada Packages Provide The Necessary Facility For Implementing Abstract Data Types

Visible Part Of Package Specification

The Abstract Data Type Is Defined Here

1. A Private Type Definition Names The Type

2. Subprogram Definitions Define The Operations, Ada Allows Functions Names To Be Operator Symbols

3. Comments Can Define The Axioms, The Behavior Of The Data Type

Private Part Of Package Specification

The Representation Of The Data Type Is Defined Here As A Type Definition

Package Body

The Implementation Of The Data Type Is Defined Here As The Bodies Of The Subprograms Declared In The Specification

Important Point:

1. Only The Visible Part Of The Package Can Be Seen By Users Of The Package

# PACKAGES

## Package Syntax

package_specification ::=
  PACKAGE identifier IS
    {basic_declarative_item}
  [PRIVATE
    {basic_declarative_item}]
  END [*package*_simple_name];

package_body ::=
  PACKAGE BODY *package*_simple_name IS
    [declarative_part]
  END [*package*_simple_name];

## General Package Structure

```
PACKAGE Data_Type_Package IS
  TYPE Data_Type IS PRIVATE;
  -- Subprogram Specifications
PRIVATE
  TYPE Data_Type IS
    --Actual Type Definition
END Data_Type_Package;

PACKAGE Data_Type_Package IS
  --Subprogram Bodies
END Data_Type_Package;
```

# COMPLEX PACKAGE SPECIFICATION

## Package Specification

```
PACKAGE Complex_Package IS
  TYPE Complex IS PRIVATE;
  FUNCTION "+"(X,Y: COMPLEX) RETURN Complex;
  FUNCTION "-"(X,Y: COMPLEX) RETURN Complex;
  FUNCTION "*"(X,Y: COMPLEX) RETURN Complex;
  FUNCTION "/"(X,Y: COMPLEX) RETURN Complex;
  FUNCTION "+"(X: Float; Y: Complex) RETURN
    Complex;
  FUNCTION "*"(X: Float; Y: Complex) RETURN
    Complex;
  FUNCTION Re(X:Complex) RETURN Float;
  FUNCTION Im(X:Complex) RETURN Float;
  i: CONSTANT Complex;
PRIVATE
  TYPE Complex IS
    RECORD
      Real: Float;
      Imaginary: Float;
    END RECORD;
  i: CONSTANT Complex := (0.0,1.0);
END Complex_Package;
```

## Important Points:

1. The Second + And * Functions Are Constructors
2. The Functions Re And Im Are Selectors

# COMPLEX PACKAGE BODY

## Package Body

```
PACKAGE BODY Complex_Package IS
  FUNCTION "+"(X,Y: COMPLEX)
  RETURN Complex IS
  BEGIN
    RETURN (X.Real + Y.Real,
      X.Imaginary + Y.Imaginary);
  END "+";
  FUNCTION "*"(X,Y: COMPLEX)
  RETURN Complex IS
  BEGIN
    RETURN (X.Real * Y.Real - X.Imaginary *
      Y.Imaginary, X.Real * Y.Imaginary +
      X.Imaginary * Y.Real);
  END "*";
  FUNCTION "+"(X: Float; Y: Complex)
  RETURN Complex IS
  BEGIN
    RETURN (X + Y.Real, Y.Imaginary);
  END "+";
  FUNCTION "*"(X: Float; Y: Complex)
  RETURN Complex IS
  BEGIN
    RETURN (X * Y.Real, X * Y.Imaginary);
  END "*";
END Complex_Package;
```

# USE OF COMPLEX NUMBER PACKAGE

Complex Numbers Procedure:

```
WITH Complex_Package; USE Complex_Package;
PROCEDURE Complex_Numbers IS
  Complex1,Complex2,Complex3: Complex;
BEGIN
  Complex1 := 3.0 + 2.0 * i;
  Complex2 :=
    (Real => 1.0, Imaginary => 2.0);
  Complex3 := Complex1 + Complex2;
END Complex_Numbers;
```

Important Points:

1. In Order To Use The Complex Operators In Their Infix Form, It Is Necessary To "Use" The Package

2. Complex Numbers Are Created With The Selectors

   Using +,* (FLOAT, COMPLEXES → COMPLEXES)

   For Example, `3.0 + 2.0 * i`

3. The Second Assignment Is A Syntax Error Because The The Type Is Private

4. The Third Assignment Performs Complex Number Addition

# SOFTWARE ENGINEERING CONCEPTS

Terminology:

Encapsulation: Physically Enclosing The Type Definition And Operations For An Abstract Data Type

Information Hiding: Limiting Visibility Of Type And Variable Declarations

Loosely Coupled Systems: Systems Subdivided Into Components With Minimal Interdependence

Software Reusability: General Software That Can Be Used By Many Programs

Important Points:

1. If The Complex Number Package Were Changed From Rectangular Coordinates To Polar Coordinates Programs That Use The Package

    a) Would Require Recompilation
    b) Would Not Require Any Changes

2. Loose Coupling Minimizes The Possibility That Software Changes Will Create Problems In Other Parts Of The System

3. Ada Packages Enable Encapsulation, Information Hiding And Make It Possible To Create Loosely Coupled Systems

# CHARACTER PACKAGE SPECIFICATION

## Package Specification

```
PACKAGE Character_Function_Package IS
  FUNCTION Is_Upper(Ch: Character)
    RETURN Boolean;
  FUNCTION Is_Lower(Ch: Character)
    RETURN Boolean;
  FUNCTION Is_Alpha(Ch: Character)
    RETURN Boolean;
  FUNCTION Is_Digit(Ch: Character)
    RETURN Boolean;
  FUNCTION To_Upper(Ch: Character)
    RETURN Character;
  FUNCTION To_Lower(Ch: Character)
    RETURN Character;
END Character_Function_Package;
```

## Important Points:

1. This Package Does Not Define A New Type, It Collects Together A Library Of Similar Utility Functions
2. Packages Can Also Be Used To:
   a. Define A Collection Of Constants, Such Packages May Have No Bodies
   b. Define A Collection Of Variables, Although Ada Permits This, It Is A Very Poor Practice

# CHARACTER PACKAGE BODY

## Package Body

```
PACKAGE BODY Character_Function_Package IS
  FUNCTION Is_Upper(Ch: Character)
    RETURN Boolean IS
  BEGIN
    RETURN Ch IN 'A'..'Z';
  END Is_Upper;
  FUNCTION Is_Alpha(Ch: Character)
    RETURN Boolean IS
  BEGIN
    RETURN Is_Lower(Ch) OR Is_Upper(Ch);
  END Is_Alpha;
  FUNCTION To_Upper(Ch: Character)
    RETURN Character IS
    Offset: Integer := Character'Pos('A') -
      Character'Pos('a');
    Place: Integer;
  BEGIN
    IF Is_Lower(Ch) THEN
      Place := Character'Pos(Ch) + Offset;
      RETURN Character'Val(Place);
    ELSE
      RETURN Ch;
    END IF;
  END To_Upper;
  -- Several Function Bodies Are Omitted
END Character_Function_Package;
```

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 13

# ADVANCED ARRAY TYPES

# MULTIDIMENSIONAL ARRAYS

## Multi-Dimensional Array Definition Syntax

multi-dimensional_array_definition ::=
  ARRAY (discrete_range {,discrete_range}) OF type

## Multi-Dimensional Array Type Declarations

```
TYPE Square_Matrix IS ARRAY(1..10,1..10) OF
  Integer;

TYPE Rectangular_Matrix IS ARRAY(1..2,1..4)
  OF Integer;

TYPE Three_Dimensional_Array IS ARRAY
  (1..5,1..5,1..5) OF Float;
```

## Array Object Declarations

```
Square: Square_Matrix;

Rectangle: Rectangular_Matrix;

Three: Three_Dimensional_Array;
```

## Array Component Assignments

```
Matrix(1,1) := 1;

Rectangle(2,4) := 5;

Three(1,2,5) := 3.8;
```

# MULTI-DIMENSIONAL AGGREGATES

Positional Aggregate

```
Rectangle := ((1,2,3,4),(2,4,6,8));
```

Named Aggregate

```
Rectangle := (
  1 => (1 => 1, 2 => 2, 3 => 3, 4 => 4),
  2 => (1 => 2, 2 => 4, 3 => 6, 4 => 8));
```

# MULTI-DIMENSIONAL ARRAY OPERATIONS

Assignment

Types Must Match

Subscripting

A Subscript Must Be Provided For Each Dimension

Slicing

Slices Are Not Permitted On Multi-Dimensional Arrays

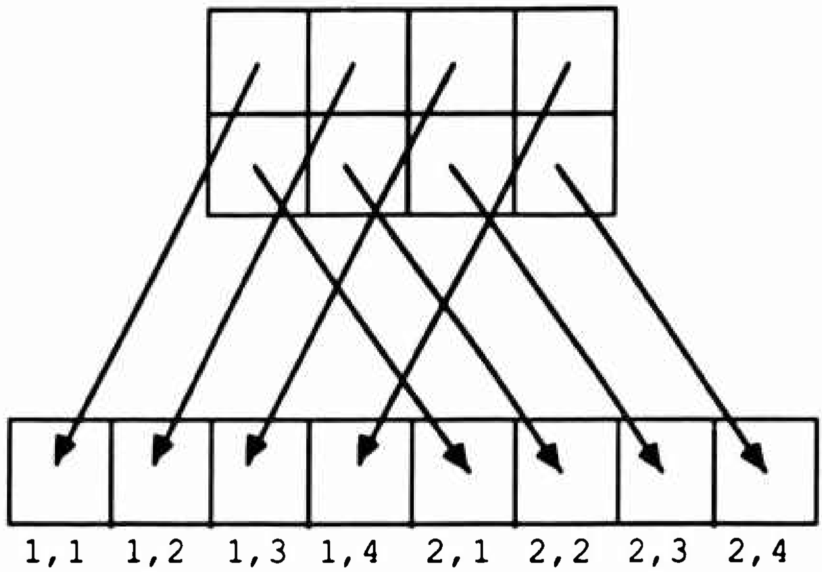
Relational Operators

Pre-defined = /=

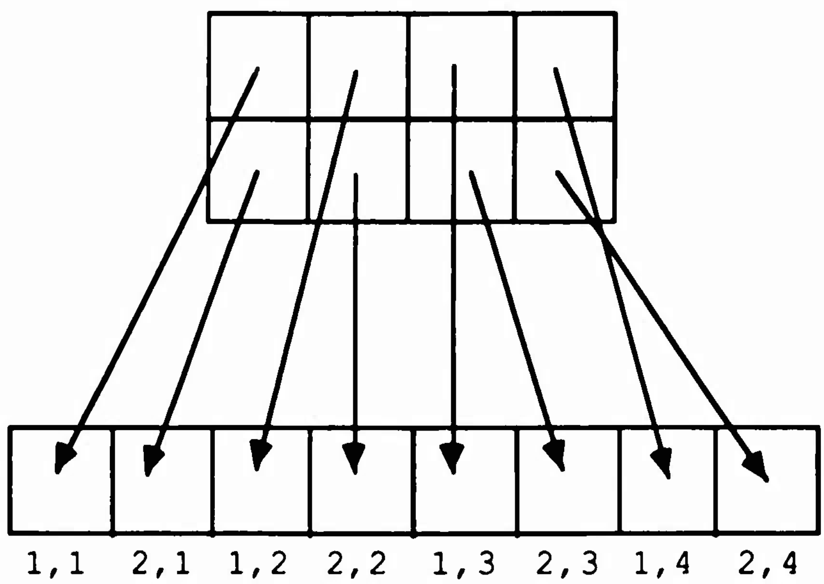
Not Pre-defined > >= < <=
May Be User Defined

# STORAGE ALLOCATION FOR ARRAYS

## Row Major




## Column Major



# TWO-DIMENSIONAL ARRAY EXAMPLE

## Type Declaration

```
SUBTYPE Row_Range IS Integer RANGE 1..2;
SUBTYPE Column_Range IS Integer RANGE 1..3;
TYPE Original_Type IS ARRAY(Row_Range,
  Column_Range) OF Integer;
TYPE Flipped_Type IS ARRAY(Column_Range,
  Row_Range) OF Integer;
```

## Matrix Transposition Function

```
FUNCTION Transpose(Original: Original_Type)
  RETURN Flipped_Type IS
  Flipped: Flipped_Type;
BEGIN
  FOR Row IN Row_Range LOOP
    FOR Column IN Column_Range LOOP
      Flipped(Column,Row) :=
        Original(Row,Column);
    END LOOP;
  END LOOP;
  RETURN Flipped;
END Transpose;
```

## Important Point:

1. A Generalized Function For Matrices Of Any Size Can Be Written Using Unconstrained Array Types

# THREE DIMENSIONAL ARRAY EXAMPLE

## Type Declarations

```
TYPE Day_Type IS (Monday, Tuesday,
  Wednesday, Thursday, Friday);
SUBTYPE Room_Type IS Integer RANGE 1..10;
TYPE Bldg_Type IS (Computer_Science,
  Engineering);
TYPE Room_Status_Type IS (Available,
  Reserved);
TYPE Room_Availability_Matrix IS
  ARRAY(Day_Type, Bldg_Type, Room_Type) OF
  Room_Status_Type;
```

## Initialized Variable Declarations

```
Room_Matrix: Room_Availability_Matrix :=
  (Monday..Friday =>
  (Computer_Science..Engineering =>
  (Room_Type'First..Room_Type'Last =>
    Available)));
```

## Constant Declaration

```
Full: CONSTANT Room_Availability_Matrix :=
  (Monday..Friday =>
  (Computer_Science..Engineering =>
  (Room_Type'First..Room_Type'Last =>
    Reserved)));
```

# UNCONSTRAINED ARRAY TYPES

## Unconstrained Array Definition Syntax

unconstrained_array_definition ::=
    ARRAY (type RANGE <> {, type RANGE <>}) OF type

## Unconstrained Array Type Declarations

```
TYPE Integer_Vector IS ARRAY
  (Integer RANGE <>) OF Integer;

TYPE Character_Frequency IS ARRAY
  (Character RANGE <>) OF Integer;
```

## Array Object Declarations

```
Vector: Integer_Vector(1..10);

Frequency: Character_Frequency('a'..'z');
```

## Important Points:

1. The Box Symbol <> Means "To Be Specified"
2. Constrained Indexes And Unconstrained Indexes Can Not Be Mixed In The Same Array Type Declaration
3. An Object Of An Unconstrained Type Must Be Constrained In The Object Declaration
4. A Formal Parameter Of An Unconstrained Array Type May Be Left Unconstrained

# ARRAY ATTRIBUTES

## Array Attributes

```
Type'First(I)
```
First Subscript Of Ith Dimension

```
Type'Last(I)
```
Last Subscript Of Ith Dimension

```
Type'Length(I)
```
Number Of Elements In Ith Dimension

```
Type'Range(I)  ⇔  Type'First(I)..Type'Last(I)
```

## Important Points:

1. Array Attributes May Be Applied To Either Array Types Or Array Objects
2. Subscript Can Be Omitted For First Dimension

## Examples

```
Vector'First -- 1
Vector'Range -- 1..10

Frequency'Last -- 'z'
Frequency'Length -- 26

Rectangle'Last(1) -- 2
Rectangle'Last(2) -- 4
```

# ASSIGNMENT AND COMPARISON

## Assignment Rule

To Assign Arrays They Must Be Of Same Type And Same Length, But The Bounds May Be Different

Assigning Arrays Of Different Lengths Produces A Constraint Error At Run Time

## Array Assignment Example

```
TYPE Array_Type IS ARRAY (Integer RANGE <>)
  OF Integer;

Array_1: Array_Type(0..9);
Array_2: Array_Type(1..10);

Array_1 := Array_2; -- Valid Assignment
```

## Comparison Rule

To Compare Arrays They Must Be Of Same Type And Same Length, But The Bounds May Be Different

Comparing Arrays Of Different Lengths Produces A Value Of False

## Array Comparison Example

```
"String" = "String "

-- False Because Lengths Are Different
```

# UNCONSTRAINED ARRAYS AND SUBPROGRAMS

Major Issues:

1. Unconstrained Array Types Decouple The Length From The Type
2. Unconstrained Array Types Make It Possible To Write Generalized Subprograms

Specific Issues:

1. Subprograms Can Have Array Parameters Of An Unconstrained Array Type, Left Unconstrained
2. Functions Can Return Arrays Of An Unconstrained Type

Storage Allocation Issues:

1. Because The Compiler Must Allocation Space For Array Variables, They Must Be Constrained In Their Declaration If Their Type Is Unconstrained
2. The Size Of An Unconstrained Formal Parameter Is Determined At Run-Time, It Depends On The Size Of The Actual Parameter

Important Point:

1. The Attributes Of An Array Parameter Are Passed With The Array Automatically, So It Is Not Necessary To Pass Them As Parameters

# UNCONSTRAINED PARAMETER EXAMPLE

## Compute Maximum Program

```
WITH Text_IO;
PROCEDURE Compute_Maximum IS
  TYPE Vectors IS ARRAY(Integer RANGE <>)
    OF Integer;
  Five_Values: Vectors(1..5);
  Maximum: Integer;
  PACKAGE Int_IO IS NEW
    Text_IO.Integer_IO(Integer);
  FUNCTION Find_Maximum(Vector:
    Vectors) RETURN Integer IS
    Local_Maximum: Integer := Integer'First;
  BEGIN
    FOR Index IN Vector'Range LOOP
      IF Vector(Index) > Local_Maximum THEN
        Local_Maximum := Vector(Index);
      END IF;
    END LOOP;
    RETURN Local_Maximum;
  END Find_Maximum;
BEGIN
  Five_Values := (20,0,89,18,43);
  Maximum := Find_Maximum(Five_Values);
  Text_IO.Put("Maximum Is ");
  Int_IO.Put(Maximum);
  Text_IO.New_Line;
END Compute_Maximum;
```

# UNCONSTRAINED FUNCTION EXAMPLE

## Reverse String Function

```
FUNCTION Reverse_String(Original: String)
RETURN String IS
  Reversed: String(Original'Range);
  Reverse_Index: Positive;
BEGIN
  FOR Original_Index IN Original'Range LOOP
    Reverse_Index := Original'Last -
      Original_Index + Original'First;
    Reversed(Reverse_Index) :=
      Original(Original_Index);
  END LOOP;
  RETURN Reversed;
END Reverse_String;
```

## Important Points:

1. String Is A Predefined Unconstrained Array Type

   ```
   TYPE String IS ARRAY(Positive RANGE <>)
     OF Character;
   ```

2. The Range Attribute Is Used To Declare A Local Variable Of The Same Size As The Formal Parameter

3. The Length Of The Returned String Is Determined By Of The Variable In The Return Statement

# CMSC 130

# INTRODUCTORY COMPUTER SCIENCE

# LECTURE 14

# RECURSION

# TRIANGULAR NUMBERS WITH ITERATION

Iterative Definition Of Triangular Numbers

$$\Delta_n = \sum_{i=1}^{n} i$$

Iterative Evaluation Of Triangular Numbers

$$\Delta_4 = \sum_{i=1}^{4} i = 1 + 2 + 3 + 4 = 10$$

Iterative Triangular Function

```
FUNCTION Triangular(Number: Positive)
RETURN Positive IS
  Sum: Integer := 0;
BEGIN
  FOR Index IN 1..Number LOOP
    Sum := Sum + Index;
  END LOOP;
  RETURN Sum;
END Triangular;
```

Important Points:

1. The Triangular Number Sequence Is The Following:

   1, 3, 6 ,10, 15, 21, 28, 36 ...

2. The Summation Symbol Of Mathematics Corresponds To The For Loop Of Ada

# TRIANGULAR NUMBERS WITH RECURSION

## Recursive Definition Of Triangular Numbers

$$\Delta_n = \begin{cases} 1 \text{ if } n = 1 \\ n + \Delta_{n-1} \text{ if } n > 1 \end{cases}$$

## Recursive Evaluation Of Triangular Numbers

$$\begin{aligned} \Delta_4 &= 4 + \Delta_3 \\ &= 4 + 3 + \Delta_2 \\ &= 4 + 3 + 2 + \Delta_1 \\ &= 4 + 3 + 2 + 1 = 10 \end{aligned}$$

## Recursive Triangular Function

```
FUNCTION Triangular(Number: Positive)
RETURN Positive IS
BEGIN
  IF Number = 1 THEN
    RETURN 1;
  ELSE
    RETURN Number + Triangular(Number-1);
  END IF;
END Triangular;
```

## Important Point:

1. The Recursive Ada Function Is Simply A Translation Of The Recursive Definition

# FACTORIAL WITH ITERATION

Iterative Definition Of Factorial

$$n! = \prod_{i=1}^{n} i$$

Iterative Evaluation Of Factorial

$$4! = \prod_{i=1}^{4} i = 1 \times 2 \times 3 \times 4 = 24$$

Iterative Triangular Function

```
FUNCTION Factorial(Number: Natural)
RETURN Positive IS
  Product: Integer := 1;
BEGIN
  FOR Index IN 1..Number LOOP
    Product := Product * Index;
  END LOOP;
  RETURN Product;
END Factorial;
```

Important Points:

1. Factorial Is The Multiplicative Analog Of The Triangular Numbers

2. The Product Symbol Of Mathematics Corresponds To The For Loop Of Ada

# FACTORIAL WITH RECURSION

## Recursive Definition Of Factorial

$$n! = \begin{cases} 1 \text{ if } n = 0 \\ n \times (n-1)! \text{ if } n > 1 \end{cases}$$

## Recursive Evaluation Of Triangular Numbers

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times 3 \times 2! \\ &= 4 \times 3 \times 2 \times 1! \\ &= 4 \times 3 \times 2 \times 1 = 24 \end{aligned}$$

## Recursive Factorial Function

```
FUNCTION Factorial(Number: Natural)
RETURN Positive IS
BEGIN
  IF Number = 0 THEN
    RETURN 1;
  ELSE
    RETURN Number * Factorial(Number-1);
  END IF;
END Factorial;
```

## Important Point:

1. Recursive Subprograms Must Contain A Base Case Path And A Recursive Case Path, The Recursive Case Must Converge Toward The Base Case

# PALINDROMES WITH ITERATION

## Iterative Definitions Of Palindrome

$$\text{Palindrome}(s) = \bigwedge_{i=1}^{\frac{n}{2}} s_i = s_{n-i+1}, \text{ where } n = \text{Length}(s)$$

$$\text{Palindrome}(s) = \forall i, 1 \leq i \leq \tfrac{n}{2} \Rightarrow s_i = s_{n-i+1}$$

( $\forall$ Is Equivalent To An Iterative $\wedge$ )

## Iterative Evaluation Of Palindrome

Palindrome(*abbcba*) = $(a = a) \wedge (b = b) \wedge (b = c)$ =
True $\wedge$ True $\wedge$ False = False

## Iterative Palindrome Function

```
FUNCTION Palindrome(Word: String)
RETURN Boolean IS
  Right: Integer;
BEGIN
  FOR Index IN Word'First..Word'Last/2 LOOP
    Right := Word'Last - Index + 1;
    IF Word(Index) /= Word(Right) THEN
      RETURN False;
    END IF;
  END LOOP;
  RETURN True;
END Palindrome;
```

# PALINDROMES WITH RECURSION

## Recursive Definition Of Palindrome

$$\text{Palindrome}(s) = \begin{cases} \text{True if Length}(s) \le 1 \\ \text{First}(s) = \text{Last}(s) \wedge \text{Palindrome}(\text{Middle}(s)) \end{cases}$$

## Recursive Evaluation Of Palindrome

$$\begin{aligned} \text{Palindrome}(abbcba) &= (a = a) \wedge \text{Palindrome}(bbcb) \\ &= \text{True} \wedge \text{Palindrome}(bbcb) \\ &= \text{True} \wedge (b = b) \wedge \text{Palindrome}(bc) \\ &= \text{True} \wedge \text{True} \wedge \text{Palindrome}(bc) \\ &= \text{True} \wedge \text{True} \wedge (b = c) \\ &= \text{True} \wedge \text{True} \wedge \text{False} = \text{False} \end{aligned}$$

## Recursive Palindrome Function

```
FUNCTION Palindrome(Word: String)
RETURN Boolean IS
BEGIN
  IF Word'Length <= 1 THEN
    RETURN True;
  ELSE
    RETURN
      Word(Word'First) = Word(Word'Last)
    AND THEN
      Palindrome(Word(Word'First+1 ..
        Word'Last-1));
  END IF;
END Palindrome;
```

# COMPARING ITERATION AND RECURSION

| | Iteration | Recursion |
|---|---|---|
| Control | Loop Statement | Recursive Call |
| Local Variables | Required | Not Required |
| Assignments | Required | Not Required |
| Style | Imperative | Declarative |
| Size | Larger | Smaller |
| Nontermination | Infinite Loop | Infinite Recursion |

Important Points:

1. Recursion Provides The Beginning Of A Functional Style Of Programming That Is Characterized By:

   a) No Local Variables
   b) No Intermediate States
   c) No Assignment Statements

2. Functional Programming Is At A Higher Level Of Abstraction Than Imperative Programming

3. Thinking Recursively Means Searching For A Definition, Not For An Algorithm

4. Recursion Can Often Provide Shorter Simpler Solutions

# TAIL RECURSION

Terminology:

<u>Tail Recursive</u>: A Subprogram In Which The Recursive Call Is The Last Step

Tail Recursive Triangular Function

```
FUNCTION Triangular(Number: Positive)
RETURN Positive IS
  FUNCTION Triangle(Number,Sum: Integer)
  RETURN Positive IS
  BEGIN
    IF Number = 0 THEN
      RETURN Sum;
    ELSE
      RETURN Triangle(Number-1,Sum+Number);
    END IF;
  END Triangle;
BEGIN
  RETURN Triangle(Number,0);
END Triangular;
```

Important Points:

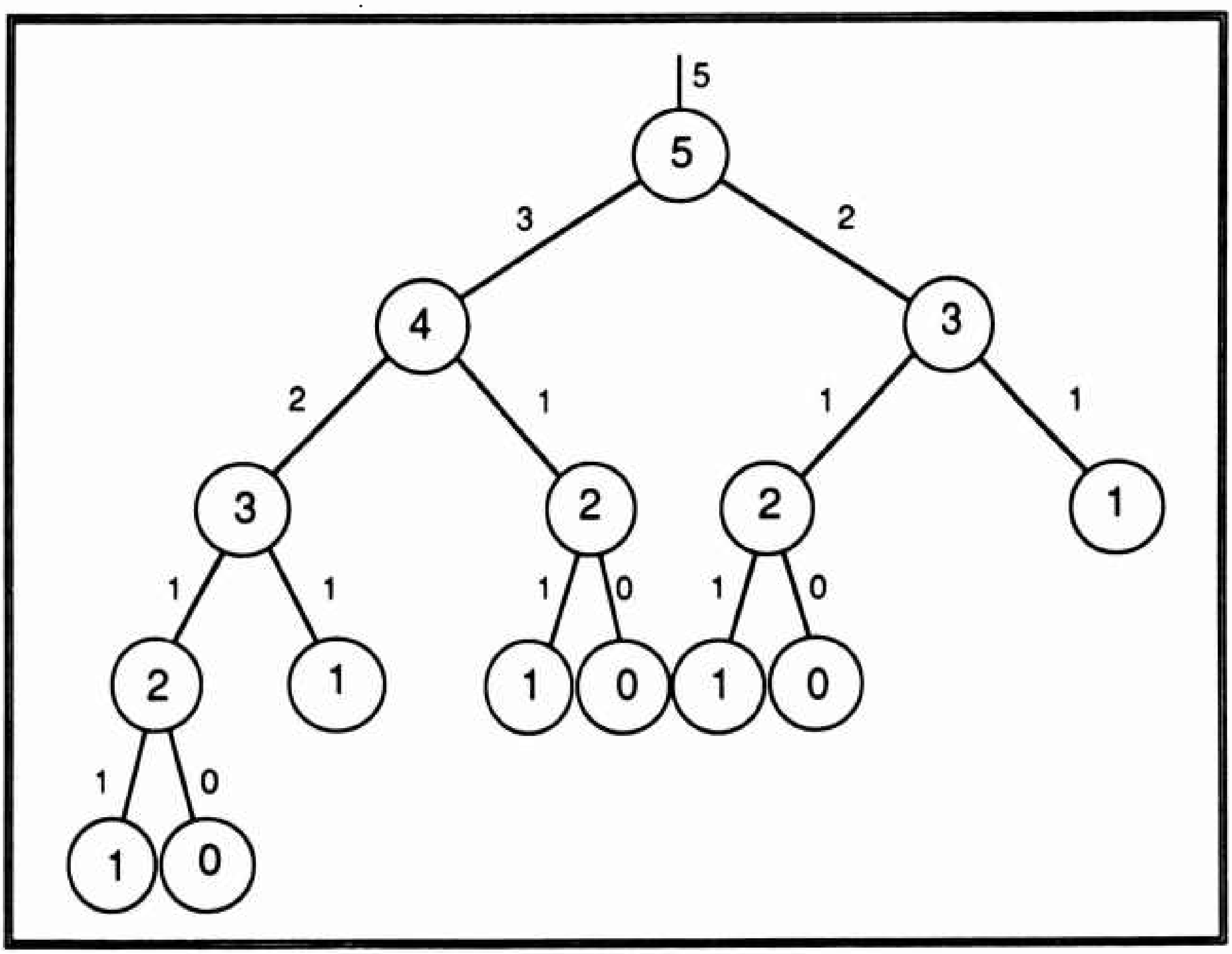
1. Tail Recursive Solutions Mimic Iteration, They Are Not The Result Of Thinking Recursively

2. The Tail Recursive Solution For This Problem Requires A Nested Function

# DOUBLE RECURSION

## Fibonacci Function

```
FUNCTION Fibonacci(Number: Natural)
RETURN Natural IS
BEGIN
  IF Number = 0 OR Number = 1 THEN
    RETURN Number;
  ELSE
    RETURN Fibonacci(Number-1) +
      Fibonacci(Number-2);
  END IF;
END Fibonacci;
```



# CHARACTER REVERSAL

## Reversal Procedure

```
WITH Text_IO;
PROCEDURE Reversal IS
  Char: Character;
BEGIN
  IF NOT Text_IO.End_Of_Line THEN
    Text_IO.Get(Char);
    Reversal;
    Text_IO.Put(Char);
  END IF;
END Reversal;
```

## Important Points:

1. This Procedure Uses The Compiler's Stack Of Activation Records To Perform The Reversal

2. The Order Of Execution Is That All The Gets Are Executed Before All The Puts

| | |
|---|---|
| $\texttt{Char}_1$ | a |
| $\texttt{Char}_2$ | b |
| $\texttt{Char}_3$ | c |
| $\texttt{Char}_4$ | d |

Local Variable Stack

# RECURSION AND EFFICIENCY

Fibonacci Numbers

| | Execution Speed | Memory Utilization |
|---|---|---|
| Recursive | Exponential | Linear |
| Iterative | Linear | Constant |

Character Reversal

| | Execution Speed | Memory Utilization |
|---|---|---|
| Recursive | Linear | Linear |
| Iterative | Linear | Linear |

Important Points:

1. The Use Of Recursion Can Dramatically Reduce The Efficiency Of Certain Problems
2. Other Problems Such As Character Reversal, Can Not Be Solved In A Bounded Memory Space, For Such Problems Recursion Does Not Introduce An Efficiency Penalty

# REFERENCES AND NOTES

## REFERENCES:

The following references were used in the preparation of these lecture notes:

*Reference Manual For The Ada Programming Language,* ANSI/MIL-STD 1815A-1983. Washington, D.C.: U.S. Government, 1983.

Barnes, J. G. P. *Programming In Ada, Third Edition.* Reading, Mass.: Addison-Wesley, 1989.

Cohen, Norman H. *Ada As A Second Language.* New York: McGraw Hill, 1986.

Feldman, Michael B. and Elliot B. Koffman. *Ada Problem Solving and Program Design.* Reading, Mass.: Addison-Wesley, 1992.

Savitch, Walter J. and Charles G. Petersen. *Ada, An Introduction to the Art and Science of Programming.* Redwood City, Calif.: Benjamin/Cummings Publishing Company, 1992.

Sebesta, Robert W. *Concepts of Programming Languages.* Redwood City, Calif.: Benjamin/Cummings, 1989.

Shumate, Ken. *Understanding Ada (2nd Edition).* New York: John Wiley, 1989.

Skansholm, Jan. *Ada from the Beginning.* Reading, Mass.: Addison-Wesley, 1988.

Volper, Dennis and Martin D. Katz. *Introduction to Programming Using Ada.* Englewood Cliffs, N.J.: Prentice-Hall, 1990.

## NOTES:

1. The Ada language reference manual and references by Cohen and Barnes provide a comprehensive view of the Ada language.

2. The reference by Sebesta provides a good discussion of programming language semantics.

3. The remaining references are all texts for introductory programming courses using Ada.